Eighth Edition

Software Engineering

A PRACTITIONER'S APPROACH

Roger S.
PRESSMAN

Bruce R.
MAXIM

Software Engineering

A PRACTITIONER'S APPROACH

This page intentionally left blank

Software Engineering

A PRACTITIONER'S APPROACH

EIGHTH EDITION

Roger S. Pressman, Ph.D.
Bruce R. Maxim, Ph.D.

McGraw Hill Education

SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, EIGHTH EDITION

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright © 2015 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. Previous editions © 2010, 2005, and 2001. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw-Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 1 0 9 8 7 6 5 4

ISBN 978-0-07-802212-8
MHID 0-07-802212-6

Senior Vice President, Products & Markets: *Kurt L. Strand*
Vice President, General Manager: *Marty Lange*
Vice President, Content Production & Technology Services: *Kimberly Meriwether David*
Managing Director: *Thomas Timp*
Publisher: *Raghu Srinivasan*
Developmental Editor: *Vincent Bradshaw*
Marketing Manager: *Heather Wagner*
Director, Content Production: *Terri Schiesl*
Project Manager: *Heather Ervolino*
Buyer: *Sandy Ludovissy*
Cover Designer: *Studio Montage, St. Louis, MO.*
Cover Image: *Farinaz Taghavi/Getty images*
Compositor: *MPS Limited*
Typeface: *8.5/13.5 Impressum Std*
Printer: *R. R. Donnelley*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Pressman, Roger S.
Software engineering : a practitioner's approach / Roger S. Pressman, Ph.D. — Eighth edition.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-07-802212-8 (alk. paper)
ISBN-10: 0-07-802212-6 (alk. paper)
1. Software engineering. I. Title.
QA76.758.P75 2015
005.1—dc23

2013035493

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw-Hill Education, and McGraw-Hill Education does not guarantee the accuracy of the information presented at these sites.

www.mhhe.com

To my granddaughters Lily and Maya, who already understand the importance of software, even though they're still in preschool.

—Roger S. Pressman

In loving memory of my parents, who taught me from an early age that pursuing a good education was far more important than pursuing money.

—Bruce R. Maxim

About the Authors

Roger S. Pressman is an internationally recognized consultant and author in software engineering. For more than four decades, he has worked as a software engineer, a manager, a professor, an author, a consultant, and an entrepreneur.

Dr. Pressman is president of R. S. Pressman & Associates, Inc., a consulting firm that specializes in helping companies establish effective software engineering practices. Over the years he has developed a set of techniques and tools that improve software engineering practice. He is also the founder of Teslaccessories, LLC, a start-up manufacturing company that specializes in custom products for the Tesla Model S electric vehicle.

Dr. Pressman is the author of nine books, including two novels, and many technical and management papers. He has been on the editorial boards of *IEEE Software* and *The Cutter IT Journal* and was editor of the "Manager" column in *IEEE Software*.

Dr. Pressman is a well-known speaker, keynoting a number of major industry conferences. He has presented tutorials at the International Conference on Software Engineering and at many other industry meetings. He has been a member of the ACM, IEEE, and Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu, and Pi Tau Sigma.

Bruce R. Maxim has worked as a software engineer, project manager, professor, author, and consultant for more than thirty years. His research interests include software engineering, human computer interaction, game design, social media, artificial intelligence, and computer science education.

Dr. Maxim is associate professor of computer and information science at the University of Michigan—Dearborn. He established the GAME Lab in the College of Engineering and Computer Science. He has published a number of papers on computer algorithm animation, game development, and engineering education. He is coauthor of a best-selling introductory computer science text. Dr. Maxim has supervised several hundred industry-based software development projects as part of his work at UM-Dearborn.

Dr. Maxim's professional experience includes managing research information systems at a medical school, directing instructional computing for a medical campus, and working as a statistical programmer. Dr. Maxim served as the chief technology officer for a game development company.

Dr. Maxim was the recipient of several distinguished teaching awards and a distinguished community service award. He is a member of Sigma Xi, Upsilon Pi Epsilon, Pi Mu Epsilon, Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers, and International Game Developers Association.

Contents at a Glance

CHAPTER 1 The Nature of Software 1
CHAPTER 2 Software Engineering 14

PART ONE **THE SOFTWARE PROCESS** 29

CHAPTER 3 Software Process Structure 30
CHAPTER 4 Process Models 40
CHAPTER 5 Agile Development 66
CHAPTER 6 Human Aspects of Software Engineering 87

PART TWO **MODELING** 103

CHAPTER 7 Principles That Guide Practice 104
CHAPTER 8 Understanding Requirements 131
CHAPTER 9 Requirements Modeling: Scenario-Based Methods 166
CHAPTER 10 Requirements Modeling: Class-Based Methods 184
CHAPTER 11 Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps 202
CHAPTER 12 Design Concepts 224
CHAPTER 13 Architectural Design 252
CHAPTER 14 Component-Level Design 285
CHAPTER 15 User Interface Design 317
CHAPTER 16 Pattern-Based Design 347
CHAPTER 17 WebApp Design 371
CHAPTER 18 MobileApp Design 391

PART THREE **QUALITY MANAGEMENT** 411

CHAPTER 19 Quality Concepts 412
CHAPTER 20 Review Techniques 431
CHAPTER 21 Software Quality Assurance 448
CHAPTER 22 Software Testing Strategies 466
CHAPTER 23 Testing Conventional Applications 496
CHAPTER 24 Testing Object-Oriented Applications 523
CHAPTER 25 Testing Web Applications 540
CHAPTER 26 Testing MobileApps 567

CHAPTER 27 Security Engineering 584
CHAPTER 28 Formal Modeling and Verification 601
CHAPTER 29 Software Configuration Management 623
CHAPTER 30 Product Metrics 653

PART FOUR MANAGING SOFTWARE PROJECTS 683

CHAPTER 31 Project Management Concepts 684
CHAPTER 32 Process and Project Metrics 703
CHAPTER 33 Estimation for Software Projects 727
CHAPTER 34 Project Scheduling 754
CHAPTER 35 Risk Management 777
CHAPTER 36 Maintenance and Reengineering 795

PART FIVE ADVANCED TOPICS 817

CHAPTER 37 Software Process Improvement 818
CHAPTER 38 Emerging Trends in Software Engineering 839
CHAPTER 39 Concluding Comments 860

APPENDIX 1 An Introduction to UML 869
APPENDIX 2 Object-Oriented Concepts 891
APPENDIX 3 Formal Methods 899
REFERENCES 909
INDEX 933

Table of Contents

Preface *xxvii*

CHAPTER 1 THE NATURE OF SOFTWARE 1

1.1 The Nature of Software 3
- 1.1.1 Defining Software 4
- 1.1.2 Software Application Domains 6
- 1.1.3 Legacy Software 7

1.2 The Changing Nature of Software 9
- 1.2.1 WebApps 9
- 1.2.2 Mobile Applications 9
- 1.2.3 Cloud Computing 10
- 1.2.4 Product Line Software 11

1.3 Summary 11

PROBLEMS AND POINTS TO PONDER 12

FURTHER READINGS AND INFORMATION SOURCES 12

CHAPTER 2 SOFTWARE ENGINEERING 14

2.1 Defining the Discipline 15

2.2 The Software Process 16
- 2.2.1 The Process Framework 17
- 2.2.2 Umbrella Activities 18
- 2.2.3 Process Adaptation 18

2.3 Software Engineering Practice 19
- 2.3.1 The Essence of Practice 19
- 2.3.2 General Principles 21

2.4 Software Development Myths 23

2.5 How It All Starts 26

2.6 Summary 27

PROBLEMS AND POINTS TO PONDER 27

FURTHER READINGS AND INFORMATION SOURCES 27

PART ONE THE SOFTWARE PROCESS 29

CHAPTER 3 SOFTWARE PROCESS STRUCTURE 30

3.1 A Generic Process Model 31

3.2 Defining a Framework Activity 32

3.3 Identifying a Task Set 34

3.4 Process Patterns 35

3.5 Process Assessment and Improvement 37

3.6 Summary 38

PROBLEMS AND POINTS TO PONDER 38

FURTHER READINGS AND INFORMATION SOURCES 39

CHAPTER 4 PROCESS MODELS 40

4.1 Prescriptive Process Models 41
- 4.1.1 The Waterfall Model 41
- 4.1.2 Incremental Process Models 43
- 4.1.3 Evolutionary Process Models 45
- 4.1.4 Concurrent Models 49
- 4.1.5 A Final Word on Evolutionary Processes 51

4.2 Specialized Process Models 52
- 4.2.1 Component-Based Development 53
- 4.2.2 The Formal Methods Model 53
- 4.2.3 Aspect-Oriented Software Development 54

4.3 The Unified Process 55
- 4.3.1 A Brief History 56
- 4.3.2 Phases of the Unified Process 56

4.4 Personal and Team Process Models 59
- 4.4.1 Personal Software Process 59
- 4.4.2 Team Software Process 60

4.5 Process Technology 61

4.6 Product and Process 62

4.7 Summary 64

PROBLEMS AND POINTS TO PONDER 64

FURTHER READINGS AND INFORMATION SOURCES 65

CHAPTER 5 AGILE DEVELOPMENT 66

5.1 What Is Agility? 68

5.2 Agility and the Cost of Change 68

5.3 What Is an Agile Process? 69
- 5.3.1 Agility Principles 70
- 5.3.2 The Politics of Agile Development 71

5.4 Extreme Programming 72
- 5.4.1 The XP Process 72
- 5.4.2 Industrial XP 75

5.5 Other Agile Process Models 77
- 5.5.1 Scrum 78
- 5.5.2 Dynamic Systems Development Method 79
- 5.5.3 Agile Modeling 80
- 5.5.4 Agile Unified Process 82

5.6 A Tool Set for the Agile Process 83

5.7 Summary 84

PROBLEMS AND POINTS TO PONDER 85

FURTHER READINGS AND INFORMATION SOURCES 85

CHAPTER 6 HUMAN ASPECTS OF SOFTWARE ENGINEERING 87

6.1 Characteristics of a Software Engineer 88

6.2 The Psychology of Software Engineering 89

6.3 The Software Team 90

6.4 Team Structures 92

6.5 Agile Teams 93
- 6.5.1 The Generic Agile Team 93
- 6.5.2 The XP Team 94

6.6 The Impact of Social Media 95
6.7 Software Engineering Using the Cloud 97
6.8 Collaboration Tools 98
6.9 Global Teams 99
6.10 Summary 100
PROBLEMS AND POINTS TO PONDER 101
FURTHER READINGS AND INFORMATION SOURCES 102

PART TWO MODELING 103

CHAPTER 7 PRINCIPLES THAT GUIDE PRACTICE 104

7.1 Software Engineering Knowledge 105
7.2 Core Principles 106
7.2.1 Principles That Guide Process 106
7.2.2 Principles That Guide Practice 107
7.3 Principles That Guide Each Framework Activity 109
7.3.1 Communication Principles 110
7.3.2 Planning Principles 112
7.3.3 Modeling Principles 114
7.3.4 Construction Principles 121
7.3.5 Deployment Principles 125
7.4 Work Practices 126
7.5 Summary 127
PROBLEMS AND POINTS TO PONDER 128
FURTHER READINGS AND INFORMATION SOURCES 129

CHAPTER 8 UNDERSTANDING REQUIREMENTS 131

8.1 Requirements Engineering 132
8.2 Establishing the Groundwork 138
8.2.1 Identifying Stakeholders 139
8.2.2 Recognizing Multiple Viewpoints 139
8.2.3 Working toward Collaboration 140
8.2.4 Asking the First Questions 140
8.2.5 Nonfunctional Requirements 141
8.2.6 Traceability 142
8.3 Eliciting Requirements 142
8.3.1 Collaborative Requirements Gathering 143
8.3.2 Quality Function Deployment 146
8.3.3 Usage Scenarios 146
8.3.4 Elicitation Work Products 147
8.3.5 Agile Requirements Elicitation 148
8.3.6 Service-Oriented Methods 148
8.4 Developing Use Cases 149
8.5 Building the Analysis Model 154
8.5.1 Elements of the Analysis Model 154
8.5.2 Analysis Patterns 157
8.5.3 Agile Requirements Engineering 158
8.5.4 Requirements for Self-Adaptive Systems 158
8.6 Negotiating Requirements 159

8.7 Requirements Monitoring 160
8.8 Validating Requirements 161
8.9 Avoiding Common Mistakes 162
8.10 Summary 162
PROBLEMS AND POINTS TO PONDER 163
FURTHER READINGS AND OTHER INFORMATION SOURCES 164

CHAPTER 9 REQUIREMENTS MODELING: SCENARIO-BASED METHODS 166

9.1 Requirements Analysis 167
9.1.1 Overall Objectives and Philosophy 168
9.1.2 Analysis Rules of Thumb 169
9.1.3 Domain Analysis 170
9.1.4 Requirements Modeling Approaches 171
9.2 Scenario-Based Modeling 173
9.2.1 Creating a Preliminary Use Case 173
9.2.2 Refining a Preliminary Use Case 176
9.2.3 Writing a Formal Use Case 177
9.3 UML Models That Supplement the Use Case 179
9.3.1 Developing an Activity Diagram 180
9.3.2 Swimlane Diagrams 181
9.4 Summary 182
PROBLEMS AND POINTS TO PONDER 182
FURTHER READINGS AND INFORMATION SOURCES 183

CHAPTER 10 REQUIREMENTS MODELING: CLASS-BASED METHODS 184

10.1 Identifying Analysis Classes 185
10.2 Specifying Attributes 188
10.3 Defining Operations 189
10.4 Class-Responsibility-Collaborator Modeling 192
10.5 Associations and Dependencies 198
10.6 Analysis Packages 199
10.7 Summary 200
PROBLEMS AND POINTS TO PONDER 201
FURTHER READINGS AND INFORMATION SOURCES 201

CHAPTER 11 REQUIREMENTS MODELING: BEHAVIOR, PATTERNS, AND WEB/MOBILE APPS 202

11.1 Creating a Behavioral Model 203
11.2 Identifying Events with the Use Case 203
11.3 State Representations 204
11.4 Patterns for Requirements Modeling 207
11.4.1 Discovering Analysis Patterns 208
11.4.2 A Requirements Pattern Example: Actuator-Sensor 209
11.5 Requirements Modeling for Web and Mobile Apps 213
11.5.1 How Much Analysis Is Enough? 214
11.5.2 Requirements Modeling Input 214
11.5.3 Requirements Modeling Output 215
11.5.4 Content Model 216

11.5.5 Interaction Model for Web and Mobile Apps 217
11.5.6 Functional Model 218
11.5.7 Configuration Models for WebApps 219
11.5.8 Navigation Modeling 220
11.6 Summary 221
PROBLEMS AND POINTS TO PONDER 222
FURTHER READINGS AND INFORMATION SOURCES 222

CHAPTER 12 DESIGN CONCEPTS 224

12.1 Design within the Context of Software Engineering 225
12.2 The Design Process 228
12.2.1 Software Quality Guidelines and Attributes 228
12.2.2 The Evolution of Software Design 230
12.3 Design Concepts 231
12.3.1 Abstraction 232
12.3.2 Architecture 232
12.3.3 Patterns 233
12.3.4 Separation of Concerns 234
12.3.5 Modularity 234
12.3.6 Information Hiding 235
12.3.7 Functional Independence 236
12.3.8 Refinement 237
12.3.9 Aspects 237
12.3.10 Refactoring 238
12.3.11 Object-Oriented Design Concepts 238
12.3.12 Design Classes 239
12.3.13 Dependency Inversion 241
12.3.14 Design for Test 242
12.4 The Design Model 243
12.4.1 Data Design Elements 244
12.4.2 Architectural Design Elements 244
12.4.3 Interface Design Elements 245
12.4.4 Component-Level Design Elements 247
12.4.5 Deployment-Level Design Elements 248
12.5 Summary 249
PROBLEMS AND POINTS TO PONDER 250
FURTHER READINGS AND INFORMATION SOURCES 251

CHAPTER 13 ARCHITECTURAL DESIGN 252

13.1 Software Architecture 253
13.1.1 What Is Architecture? 253
13.1.2 Why Is Architecture Important? 254
13.1.3 Architectural Descriptions 255
13.1.4 Architectural Decisions 256
13.2 Architectural Genres 257
13.3 Architectural Styles 258
13.3.1 A Brief Taxonomy of Architectural Styles 258
13.3.2 Architectural Patterns 263
13.3.3 Organization and Refinement 263
13.4 Architectural Considerations 264

13.5 Architectural Decisions 266
13.6 Architectural Design 267
13.6.1 Representing the System in Context 267
13.6.2 Defining Archetypes 269
13.6.3 Refining the Architecture into Components 270
13.6.4 Describing Instantiations of the System 272
13.6.5 Architectural Design for Web Apps 273
13.6.6 Architectural Design for Mobile Apps 274
13.7 Assessing Alternative Architectural Designs 274
13.7.1 Architectural Description Languages 276
13.7.2 Architectural Reviews 277
13.8 Lessons Learned 278
13.9 Pattern-based Architecture Review 278
13.10 Architecture Conformance Checking 279
13.11 Agility and Architecture 280
13.12 Summary 282
PROBLEMS AND POINTS TO PONDER 282
FURTHER READINGS AND INFORMATION SOURCES 283

CHAPTER 14 COMPONENT-LEVEL DESIGN 285

14.1 What Is a Component? 286
14.1.1 An Object-Oriented View 286
14.1.2 The Traditional View 288
14.1.3 A Process-Related View 291
14.2 Designing Class-Based Components 291
14.2.1 Basic Design Principles 292
14.2.2 Component-Level Design Guidelines 295
14.2.3 Cohesion 296
14.2.4 Coupling 298
14.3 Conducting Component-Level Design 299
14.4 Component-Level Design for WebApps 305
14.4.1 Content Design at the Component Level 306
14.4.2 Functional Design at the Component Level 306
14.5 Component-Level Design for Mobile Apps 306
14.6 Designing Traditional Components 307
14.7 Component-Based Development 308
14.7.1 Domain Engineering 308
14.7.2 Component Qualification, Adaptation, and Composition 309
14.7.3 Architectural Mismatch 311
14.7.4 Analysis and Design for Reuse 312
14.7.5 Classifying and Retrieving Components 312
14.8 Summary 313
PROBLEMS AND POINTS TO PONDER 315
FURTHER READINGS AND INFORMATION SOURCES 315

CHAPTER 15 USER INTERFACE DESIGN 317

15.1 The Golden Rules 318
15.1.1 Place the User in Control 318
15.1.2 Reduce the User's Memory Load 319
15.1.3 Make the Interface Consistent 321

15.2 User Interface Analysis and Design 322
15.2.1 Interface Analysis and Design Models 322
15.2.2 The Process 323
15.3 Interface Analysis 325
15.3.1 User Analysis 325
15.3.2 Task Analysis and Modeling 326
15.3.3 Analysis of Display Content 331
15.3.4 Analysis of the Work Environment 331
15.4 Interface Design Steps 332
15.4.1 Applying Interface Design Steps 332
15.4.2 User Interface Design Patterns 334
15.4.3 Design Issues 335
15.5 WebApp and Mobile Interface Design 337
15.5.1 Interface Design Principles and Guidelines 337
15.5.2 Interface Design Workflow for Web and Mobile Apps 341
15.6 Design Evaluation 342
15.7 Summary 344
PROBLEMS AND POINTS TO PONDER 345
FURTHER READINGS AND INFORMATION SOURCES 346

CHAPTER 16 PATTERN-BASED DESIGN 347

16.1 Design Patterns 348
16.1.1 Kinds of Patterns 349
16.1.2 Frameworks 351
16.1.3 Describing a Pattern 352
16.1.4 Pattern Languages and Repositories 353
16.2 Pattern-Based Software Design 354
16.2.1 Pattern-Based Design in Context 354
16.2.2 Thinking in Patterns 354
16.2.3 Design Tasks 356
16.2.4 Building a Pattern-Organizing Table 358
16.2.5 Common Design Mistakes 359
16.3 Architectural Patterns 359
16.4 Component-Level Design Patterns 360
16.5 User Interface Design Patterns 362
16.6 WebApp Design Patterns 364
16.6.1 Design Focus 365
16.6.2 Design Granularity 365
16.7 Patterns for Mobile Apps 366
16.8 Summary 367
PROBLEMS AND POINTS TO PONDER 368
FURTHER READINGS AND INFORMATION SOURCES 369

CHAPTER 17 WEBAPP DESIGN 371

17.1 WebApp Design Quality 372
17.2 Design Goals 374
17.3 A Design Pyramid for WebApps 375
17.4 WebApp Interface Design 376

17.5 Aesthetic Design 377
17.5.1 Layout Issues 378
17.5.2 Graphic Design Issues 378
17.6 Content Design 379
17.6.1 Content Objects 379
17.6.2 Content Design Issues 380
17.7 Architecture Design 381
17.7.1 Content Architecture 381
17.7.2 WebApp Architecture 384
17.8 Navigation Design 385
17.8.1 Navigation Semantics 385
17.8.2 Navigation Syntax 387
17.9 Component-Level Design 387
17.10 Summary 388
PROBLEMS AND POINTS TO PONDER 389
FURTHER READINGS AND INFORMATION SOURCES 389

CHAPTER 18 MOBILEAPP DESIGN 391

18.1 The Challenges 392
18.1.1 Development Considerations 392
18.1.2 Technical Considerations 393
18.2 Developing MobileApps 395
18.2.1 MobileApp Quality 397
18.2.2 User Interface Design 398
18.2.3 Context-Aware Apps 399
18.2.4 Lessons Learned 400
18.3 MobileApp Design—Best Practices 401
18.4 Mobility Environments 403
18.5 The Cloud 405
18.6 The Applicability of Conventional Software Engineering 407
18.7 Summary 408
PROBLEMS AND POINTS TO PONDER 409
FURTHER READINGS AND INFORMATION SOURCES 409

PART THREE QUALITY MANAGEMENT 411

CHAPTER 19 QUALITY CONCEPTS 412

19.1 What Is Quality? 413
19.2 Software Quality 414
19.2.1 Garvin's Quality Dimensions 415
19.2.2 McCall's Quality Factors 416
19.2.3 ISO 9126 Quality Factors 418
19.2.4 Targeted Quality Factors 418
19.2.5 The Transition to a Quantitative View 420
19.3 The Software Quality Dilemma 420
19.3.1 "Good Enough" Software 421
19.3.2 The Cost of Quality 422
19.3.3 Risks 424
19.3.4 Negligence and Liability 425

19.3.5 Quality and Security 425
19.3.6 The Impact of Management Actions 426
19.4 Achieving Software Quality 427
19.4.1 Software Engineering Methods 427
19.4.2 Project Management Techniques 427
19.4.3 Quality Control 427
19.4.4 Quality Assurance 428
19.5 Summary 428
PROBLEMS AND POINTS TO PONDER 429
FURTHER READINGS AND INFORMATION SOURCES 429

CHAPTER 20 REVIEW TECHNIQUES 431

20.1 Cost Impact of Software Defects 432
20.2 Defect Amplification and Removal 433
20.3 Review Metrics and Their Use 435
20.3.1 Analyzing Metrics 435
20.3.2 Cost-Effectiveness of Reviews 436
20.4 Reviews: A Formality Spectrum 438
20.5 Informal Reviews 439
20.6 Formal Technical Reviews 441
20.6.1 The Review Meeting 441
20.6.2 Review Reporting and Record Keeping 442
20.6.3 Review Guidelines 442
20.6.4 Sample-Driven Reviews 444
20.7 Post-Mortem Evaluations 445
20.8 Summary 446
PROBLEMS AND POINTS TO PONDER 446
FURTHER READINGS AND INFORMATION SOURCES 447

CHAPTER 21 SOFTWARE QUALITY ASSURANCE 448

21.1 Background Issues 449
21.2 Elements of Software Quality Assurance 450
21.3 SQA Processes and Product Characteristics 452
21.4 SQA Tasks, Goals, and Metrics 452
21.4.1 SQA Tasks 453
21.4.2 Goals, Attributes, and Metrics 454
21.5 Formal Approaches to SQA 456
21.6 Statistical Software Quality Assurance 456
21.6.1 A Generic Example 457
21.6.2 Six Sigma for Software Engineering 458
21.7 Software Reliability 459
21.7.1 Measures of Reliability and Availability 459
21.7.2 Software Safety 460
21.8 The ISO 9000 Quality Standards 461
21.9 The SQA Plan 463
21.10 Summary 463
PROBLEMS AND POINTS TO PONDER 464
FURTHER READINGS AND INFORMATION SOURCES 464

CHAPTER 22 SOFTWARE TESTING STRATEGIES 466

22.1 A Strategic Approach to Software Testing 466
22.1.1 Verification and Validation 468
22.1.2 Organizing for Software Testing 468
22.1.3 Software Testing Strategy—The Big Picture 469
22.1.4 Criteria for Completion of Testing 472
22.2 Strategic Issues 472
22.3 Test Strategies for Conventional Software 473
22.3.1 Unit Testing 473
22.3.2 Integration Testing 475
22.4 Test Strategies for Object-Oriented Software 481
22.4.1 Unit Testing in the OO Context 481
22.4.2 Integration Testing in the OO Context 481
22.5 Test Strategies for WebApps 482
22.6 Test Strategies for MobileApps 483
22.7 Validation Testing 483
22.7.1 Validation-Test Criteria 484
22.7.2 Configuration Review 484
22.7.3 Alpha and Beta Testing 484
22.8 System Testing 486
22.8.1 Recovery Testing 486
22.8.2 Security Testing 486
22.8.3 Stress Testing 487
22.8.4 Performance Testing 487
22.8.5 Deployment Testing 487
22.9 The Art of Debugging 488
22.9.1 The Debugging Process 488
22.9.2 Psychological Considerations 490
22.9.3 Debugging Strategies 491
22.9.4 Correcting the Error 492
22.10 Summary 493
PROBLEMS AND POINTS TO PONDER 493
FURTHER READINGS AND INFORMATION SOURCES 494

CHAPTER 23 TESTING CONVENTIONAL APPLICATIONS 496

23.1 Software Testing Fundamentals 497
23.2 Internal and External Views of Testing 499
23.3 White-Box Testing 500
23.4 Basis Path Testing 500
23.4.1 Flow Graph Notation 500
23.4.2 Independent Program Paths 502
23.4.3 Deriving Test Cases 504
23.4.4 Graph Matrices 506
23.5 Control Structure Testing 507
23.6 Black-Box Testing 509
23.6.1 Graph-Based Testing Methods 509
23.6.2 Equivalence Partitioning 511
23.6.3 Boundary Value Analysis 512
23.6.4 Orthogonal Array Testing 513

23.7 Model-Based Testing 516
23.8 Testing Documentation and Help Facilities 516
23.9 Testing for Real-Time Systems 517
23.10 Patterns for Software Testing 519
23.11 Summary 520
PROBLEMS AND POINTS TO PONDER 521
FURTHER READINGS AND INFORMATION SOURCES 521

CHAPTER 24 TESTING OBJECT-ORIENTED APPLICATIONS 523

24.1 Broadening the View of Testing 524
24.2 Testing OOA and OOD Models 525
24.2.1 Correctness of OOA and OOD Models 525
24.2.2 Consistency of Object-Oriented Models 526
24.3 Object-Oriented Testing Strategies 528
24.3.1 Unit Testing in the OO Context 528
24.3.2 Integration Testing in the OO Context 529
24.3.3 Validation Testing in an OO Context 529
24.4 Object-Oriented Testing Methods 529
24.4.1 The Test-Case Design Implications of OO Concepts 530
24.4.2 Applicability of Conventional Test-Case Design Methods 531
24.4.3 Fault-Based Testing 531
24.4.4 Scenario-Based Test Design 532
24.5 Testing Methods Applicable at the Class Level 532
24.5.1 Random Testing for OO Classes 532
24.5.2 Partition Testing at the Class Level 533
24.6 Interclass Test-Case Design 534
24.6.1 Multiple Class Testing 534
24.6.2 Tests Derived from Behavior Models 536
24.7 Summary 537
PROBLEMS AND POINTS TO PONDER 538
FURTHER READINGS AND INFORMATION SOURCES 538

CHAPTER 25 TESTING WEB APPLICATIONS 540

25.1 Testing Concepts for WebApps 541
25.1.1 Dimensions of Quality 541
25.1.2 Errors within a WebApp Environment 542
25.1.3 Testing Strategy 543
25.1.4 Test Planning 543
25.2 The Testing Process—An Overview 544
25.3 Content Testing 545
25.3.1 Content Testing Objectives 545
25.3.2 Database Testing 547
25.4 User Interface Testing 549
25.4.1 Interface Testing Strategy 549
25.4.2 Testing Interface Mechanisms 550
25.4.3 Testing Interface Semantics 552
25.4.4 Usability Tests 552
25.4.5 Compatibility Tests 554
25.5 Component-Level Testing 555

25.6 Navigation Testing 556
25.6.1 Testing Navigation Syntax 556
25.6.2 Testing Navigation Semantics 556
25.7 Configuration Testing 558
25.7.1 Server-Side Issues 558
25.7.2 Client-Side Issues 559
25.8 Security Testing 559
25.9 Performance Testing 560
25.9.1 Performance Testing Objectives 561
25.9.2 Load Testing 562
25.9.3 Stress Testing 562
25.10 Summary 563
PROBLEMS AND POINTS TO PONDER 564
FURTHER READINGS AND INFORMATION SOURCES 565

CHAPTER 26 TESTING MOBILEAPPS 567

26.1 Testing Guidelines 568
26.2 The Testing Strategies 569
26.2.1 Are Conventional Approaches Applicable? 570
26.2.2 The Need for Automation 571
26.2.3 Building a Test Matrix 572
26.2.4 Stress Testing 573
26.2.5 Testing in a Production Environment 573
26.3 Considering the Spectrum of User Interaction 574
26.3.1 Gesture Testing 575
26.3.2 Voice Input and Recognition 576
26.3.3 Virtual Key Board Input 577
26.3.4 Alerts and Extraordinary Conditions 577
26.4 Test Across Borders 578
26.5 Real-Time Testing Issues 578
26.6 Testing Tools and Environments 579
26.7 Summary 581
PROBLEMS AND POINTS TO PONDER 582
FURTHER READINGS AND INFORMATION SOURCES 582

CHAPTER 27 SECURITY ENGINEERING 584

27.1 Analyzing Security Requirements 585
27.2 Security and Privacy in an Online World 586
27.2.1 Social Media 587
27.2.2 Mobile Applications 587
27.2.3 Cloud Computing 587
27.2.4 The Internet of Things 588
27.3 Security Engineering Analysis 588
27.3.1 Security Requirement Elicitation 589
27.3.2 Security Modeling 590
27.3.3 Measures Design 591
27.3.4 Correctness Checks 591
27.4 Security Assurance 592
27.4.1 The Security Assurance Process 592
27.4.2 Organization and Management 593

27.5 Security Risk Analysis 594
27.6 The Role of Conventional Software Engineering Activities 595
27.7 Verification of Trustworthy Systems 597
27.8 Summary 599
PROBLEMS AND POINTS TO PONDER 599
FURTHER READINGS AND INFORMATION SOURCES 600

CHAPTER 28 FORMAL MODELING AND VERIFICATION 601

28.1 The Cleanroom Strategy 602
28.2 Functional Specification 604
28.2.1 Black-Box Specification 605
28.2.2 State-Box Specification 606
28.2.3 Clear-Box Specification 607
28.3 Cleanroom Design 607
28.3.1 Design Refinement 608
28.3.2 Design Verification 608
28.4 Cleanroom Testing 610
28.4.1 Statistical Use Testing 610
28.4.2 Certification 612
28.5 Rethinking Formal Methods 612
28.6 Formal Methods Concepts 615
28.7 Alternative Arguments 618
28.8 Summary 619
PROBLEMS AND POINTS TO PONDER 620
FURTHER READINGS AND INFORMATION SOURCES 621

CHAPTER 29 SOFTWARE CONFIGURATION MANAGEMENT 623

29.1 Software Configuration Management 624
29.1.1 An SCM Scenario 625
29.1.2 Elements of a Configuration Management System 626
29.1.3 Baselines 626
29.1.4 Software Configuration Items 628
29.1.5 Management of Dependencies and Changes 628
29.2 The SCM Repository 630
29.2.1 General Features and Content 630
29.2.2 SCM Features 631
29.3 The SCM Process 632
29.3.1 Identification of Objects in the Software Configuration 633
29.3.2 Version Control 634
29.3.3 Change Control 635
29.3.4 Impact Management 638
29.3.5 Configuration Audit 639
29.3.6 Status Reporting 639
29.4 Configuration Management for Web and MobileApps 640
29.4.1 Dominant Issues 641
29.4.2 Configuration Objects 642
29.4.3 Content Management 643
29.4.4 Change Management 646
29.4.5 Version Control 648
29.4.6 Auditing and Reporting 649

29.5 Summary 650
PROBLEMS AND POINTS TO PONDER 651
FURTHER READINGS AND INFORMATION SOURCES 651

CHAPTER 30 PRODUCT METRICS 653

30.1 A Framework for Product Metrics 654
30.1.1 Measures, Metrics, and Indicators 654
30.1.2 The Challenge of Product Metrics 655
30.1.3 Measurement Principles 656
30.1.4 Goal-Oriented Software Measurement 656
30.1.5 The Attributes of Effective Software Metrics 657
30.2 Metrics for the Requirements Model 659
30.2.1 Function-Based Metrics 659
30.2.2 Metrics for Specification Quality 662
30.3 Metrics for the Design Model 663
30.3.1 Architectural Design Metrics 663
30.3.2 Metrics for Object-Oriented Design 666
30.3.3 Class-Oriented Metrics—The CK Metrics Suite 667
30.3.4 Class-Oriented Metrics—The MOOD Metrics Suite 670
30.3.5 OO Metrics Proposed by Lorenz and Kidd 671
30.3.6 Component-Level Design Metrics 671
30.3.7 Operation-Oriented Metrics 671
30.3.8 User Interface Design Metrics 672
30.4 Design Metrics for Web and Mobile Apps 672
30.5 Metrics for Source Code 675
30.6 Metrics for Testing 676
30.6.1 Halstead Metrics Applied to Testing 676
30.6.2 Metrics for Object-Oriented Testing 677
30.7 Metrics for Maintenance 678
30.8 Summary 679
PROBLEMS AND POINTS TO PONDER 679
FURTHER READINGS AND INFORMATION SOURCES 680

PART FOUR MANAGING SOFTWARE PROJECTS 683

CHAPTER 31 PROJECT MANAGEMENT CONCEPTS 684

31.1 The Management Spectrum 685
31.1.1 The People 685
31.1.2 The Product 686
31.1.3 The Process 686
31.1.4 The Project 686
31.2 People 687
31.2.1 The Stakeholders 687
31.2.2 Team Leaders 688
31.2.3 The Software Team 689
31.2.4 Agile Teams 691
31.2.5 Coordination and Communication Issues 692
31.3 The Product 693
31.3.1 Software Scope 694
31.3.2 Problem Decomposition 694

31.4 The Process 694
31.4.1 Melding the Product and the Process 695
31.4.2 Process Decomposition 696
31.5 The Project 697
31.6 The W^{5}HH Principle 698
31.7 Critical Practices 699
31.8 Summary 700
PROBLEMS AND POINTS TO PONDER 700
FURTHER READINGS AND INFORMATION SOURCES 701

CHAPTER 32 PROCESS AND PROJECT METRICS 703

32.1 Metrics in the Process and Project Domains 704
32.1.1 Process Metrics and Software Process Improvement 704
32.1.2 Project Metrics 707
32.2 Software Measurement 708
32.2.1 Size-Oriented Metrics 709
32.2.2 Function-Oriented Metrics 710
32.2.3 Reconciling LOC and FP Metrics 711
32.2.4 Object-Oriented Metrics 713
32.2.5 Use Case-Oriented Metrics 714
32.2.6 WebApp Project Metrics 714
32.3 Metrics for Software Quality 716
32.3.1 Measuring Quality 717
32.3.2 Defect Removal Efficiency 718
32.4 Integrating Metrics within the Software Process 719
32.4.1 Arguments for Software Metrics 720
32.4.2 Establishing a Baseline 720
32.4.3 Metrics Collection, Computation, and Evaluation 721
32.5 Metrics for Small Organizations 721
32.6 Establishing a Software Metrics Program 722
32.7 Summary 724
PROBLEMS AND POINTS TO PONDER 724
FURTHER READINGS AND INFORMATION SOURCES 725

CHAPTER 33 ESTIMATION FOR SOFTWARE PROJECTS 727

33.1 Observations on Estimation 728
33.2 The Project Planning Process 729
33.3 Software Scope and Feasibility 730
33.4 Resources 731
33.4.1 Human Resources 731
33.4.2 Reusable Software Resources 732
33.4.3 Environmental Resources 732
33.5 Software Project Estimation 733
33.6 Decomposition Techniques 734
33.6.1 Software Sizing 734
33.6.2 Problem-Based Estimation 735
33.6.3 An Example of LOC-Based Estimation 736
33.6.4 An Example of FP-Based Estimation 738
33.6.5 Process-Based Estimation 739
33.6.6 An Example of Process-Based Estimation 740
33.6.7 Estimation with Use Cases 740

33.6.8 An Example of Estimation Using Use Case Points 742
33.6.9 Reconciling Estimates 742
33.7 Empirical Estimation Models 743
33.7.1 The Structure of Estimation Models 744
33.7.2 The COCOMO II Model 744
33.7.3 The Software Equation 744
33.8 Estimation for Object-Oriented Projects 746
33.9 Specialized Estimation Techniques 746
33.9.1 Estimation for Agile Development 746
33.9.2 Estimation for WebApp Projects 747
33.10 The Make/Buy Decision 748
33.10.1 Creating a Decision Tree 749
33.10.2 Outsourcing 750
33.11 Summary 752
PROBLEMS AND POINTS TO PONDER 752
FURTHER READINGS AND INFORMATION SOURCES 753

CHAPTER 34 PROJECT SCHEDULING 754

34.1 Basic Concepts 755
34.2 Project Scheduling 757
34.2.1 Basic Principles 758
34.2.2 The Relationship between People and Effort 759
34.2.3 Effort Distribution 760
34.3 Defining a Task Set for the Software Project 761
34.3.1 A Task Set Example 762
34.3.2 Refinement of Major Tasks 763
34.4 Defining a Task Network 764
34.5 Scheduling 765
34.5.1 Time-Line Charts 766
34.5.2 Tracking the Schedule 767
34.5.3 Tracking Progress for an OO Project 768
34.5.4 Scheduling for WebApp and Mobile Projects 769
34.6 Earned Value Analysis 772
34.7 Summary 774
PROBLEMS AND POINTS TO PONDER 774
FURTHER READINGS AND INFORMATION SOURCES 776

CHAPTER 35 RISK MANAGEMENT 777

35.1 Reactive versus Proactive Risk Strategies 778
35.2 Software Risks 778
35.3 Risk Identification 780
35.3.1 Assessing Overall Project Risk 781
35.3.2 Risk Components and Drivers 782
35.4 Risk Projection 782
35.4.1 Developing a Risk Table 783
35.4.2 Assessing Risk Impact 785
35.5 Risk Refinement 787
35.6 Risk Mitigation, Monitoring, and Management 788
35.7 The RMMM Plan 790
35.8 Summary 792

PROBLEMS AND POINTS TO PONDER 792
FURTHER READINGS AND INFORMATION SOURCES 793

CHAPTER 36 MAINTENANCE AND REENGINEERING 795

36.1 Software Maintenance 796
36.2 Software Supportability 798
36.3 Reengineering 798
36.4 Business Process Reengineering 799
36.4.1 Business Processes 799
36.4.2 A BPR Model 800
36.5 Software Reengineering 802
36.5.1 A Software Reengineering Process Model 802
36.5.2 Software Reengineering Activities 803
36.6 Reverse Engineering 805
36.6.1 Reverse Engineering to Understand Data 807
36.6.2 Reverse Engineering to Understand Processing 807
36.6.3 Reverse Engineering User Interfaces 808
36.7 Restructuring 809
36.7.1 Code Restructuring 809
36.7.2 Data Restructuring 810
36.8 Forward Engineering 811
36.8.1 Forward Engineering for Client-Server Architectures 812
36.8.2 Forward Engineering for Object-Oriented Architectures 813
36.9 The Economics of Reengineering 813
36.10 Summary 814
PROBLEMS AND POINTS TO PONDER 815
FURTHER READINGS AND INFORMATION SOURCES 816

PART FIVE ADVANCED TOPICS 817

CHAPTER 37 SOFTWARE PROCESS IMPROVEMENT 818

37.1 What Is SPI? 819
37.1.1 Approaches to SPI 819
37.1.2 Maturity Models 821
37.1.3 Is SPI for Everyone? 822
37.2 The SPI Process 823
37.2.1 Assessment and Gap Analysis 823
37.2.2 Education and Training 825
37.2.3 Selection and Justification 825
37.2.4 Installation/Migration 826
37.2.5 Evaluation 827
37.2.6 Risk Management for SPI 827
37.3 The CMMI 828
37.4 The People CMM 832
37.5 Other SPI Frameworks 832
37.6 SPI Return on Investment 834
37.7 SPI Trends 835
37.8 Summary 836
PROBLEMS AND POINTS TO PONDER 837
FURTHER READINGS AND INFORMATION SOURCES 837

CHAPTER 38 EMERGING TRENDS IN SOFTWARE ENGINEERING 839

38.1 Technology Evolution 840
38.2 Prospects for a True Engineering Discipline 841
38.3 Observing Software Engineering Trends 842
38.4 Identifying "Soft Trends" 843
38.4.1 Managing Complexity 845
38.4.2 Open-World Software 846
38.4.3 Emergent Requirements 846
38.4.4 The Talent Mix 847
38.4.5 Software Building Blocks 847
38.4.6 Changing Perceptions of "Value" 848
38.4.7 Open Source 848
38.5 Technology Directions 849
38.5.1 Process Trends 849
38.5.2 The Grand Challenge 851
38.5.3 Collaborative Development 852
38.5.4 Requirements Engineering 852
38.5.5 Model-Driven Software Development 853
38.5.6 Postmodern Design 854
38.5.7 Test-Driven Development 854
38.6 Tools-Related Trends 855
38.7 Summary 857
PROBLEMS AND POINTS TO PONDER 857
FURTHER READINGS AND INFORMATION SOURCES 858

CHAPTER 39 CONCLUDING COMMENTS 860

39.1 The Importance of Software—Revisited 861
39.2 People and the Way They Build Systems 861
39.3 New Modes for Representing Information 862
39.4 The Long View 864
39.5 The Software Engineer's Responsibility 865
39.6 A Final Comment from RSP 867

APPENDIX 1 AN INTRODUCTION TO UML 869
APPENDIX 2 OBJECT-ORIENTED CONCEPTS 891
APPENDIX 3 FORMAL METHODS 899
REFERENCES 909
INDEX 933

PREFACE

When computer software succeeds—when it meets the needs of the people who use it, when it performs flawlessly over a long period of time, when it is easy to modify and even easier to use—it can and does change things for the better. But when software fails—when its users are dissatisfied, when it is error prone, when it is difficult to change and even harder to use—bad things can and do happen. We all want to build software that makes things better, avoiding the bad things that lurk in the shadow of failed efforts. To succeed, we need discipline when software is designed and built. We need an engineering approach.

It has been almost three and a half decades since the first edition of this book was written. During that time, software engineering has evolved from an obscure idea practiced by a relatively small number of zealots to a legitimate engineering discipline. Today, it is recognized as a subject worthy of serious research, conscientious study, and tumultuous debate. Throughout the industry, software engineer has replaced programmer as the job title of preference. Software process models, software engineering methods, and software tools have been adopted successfully across a broad spectrum of industry segments.

Although managers and practitioners alike recognize the need for a more disciplined approach to software, they continue to debate the manner in which discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service today's most advanced technologies. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers, and bad things happen. In addition, debate and controversy about the true nature of the software engineering approach continue. The status of software engineering is a study in contrasts. Attitudes have changed, progress has been made, but much remains to be done before the discipline reaches full maturity.

The eighth edition of *Software Engineering: A Practitioner's Approach* is intended to serve as a guide to a maturing engineering discipline. The eighth edition, like the seven editions that preceded it, is intended for both students and practitioners, retaining its appeal as a guide to the industry professional and a comprehensive introduction to the student at the upper-level undergraduate or first-year graduate level.

The eighth edition is considerably more than a simple update. The book has been revised and restructured to improve pedagogical flow and emphasize new and important software engineering processes and practices. In addition, we have further enhanced the popular "support system" for the book, providing a comprehensive set of student, instructor, and professional resources to complement the content of the book. These resources are presented as part of a website (www.mhhe.com/pressman) specifically designed for *Software Engineering: A Practitioner's Approach*.

The Eighth Edition. The 39 chapters of the eighth edition are organized into five parts. This organization better compartmentalizes topics and assists instructors who may not have the time to complete the entire book in one term.

Part 1, *The Process,* presents a variety of different views of software process, considering all important process models and addressing the debate between prescriptive and agile process philosophies. Part 2, *Modeling,* presents analysis and design methods with an emphasis on object-oriented techniques and UML modeling. Pattern-based design and design for Web and mobile applications are also considered. Part 3, *Quality Management,* presents the concepts, procedures, techniques, and methods that enable a software team to assess software quality, review software engineering work products, conduct SQA procedures, and apply an effective testing strategy and tactics. In addition, formal modeling and verification methods are also considered. Part 4, *Managing Software Projects,* presents topics that are relevant to those who plan, manage, and control a software development project. Part 5, *Advanced Topics,* considers software process improvement and software engineering trends. Continuing in the tradition of past editions, a series of sidebars is used throughout the book to present the trials and tribulations of a (fictional) software team and to provide supplementary materials about methods and tools that are relevant to chapter topics.

The five-part organization of the eighth edition enables an instructor to "cluster" topics based on available time and student need. An entire one-term course can be built around one or more of the five parts. A software engineering survey course would select chapters from all five parts. A software engineering course that emphasizes analysis and design would select topics from Parts 1 and 2. A testing-oriented software engineering course would select topics from Parts 1 and 3, with a brief foray into Part 2. A "management course" would stress Parts 1 and 4. By organizing the eighth edition in this way, we have attempted to provide an instructor with a number of teaching options. In every case the content of the eighth edition is complemented by the following elements of the *SEPA, 8/e Support System*.

Student Resources. A wide variety of student resources includes an extensive online learning center encompassing chapter-by-chapter study guides, practice quizzes, problem solutions, and a variety of Web-based resources including software engineering checklists, an evolving collection of "tiny tools," a comprehensive case study, work product templates, and many other resources. In addition, over 1,000 categorized *Web References* allow a student to explore software engineering in greater detail and a *Reference Library* with links to more than 500 downloadable papers provides an in-depth source of advanced software engineering information.

Instructor Resources. A broad array of instructor resources has been developed to supplement the eighth edition. These include a complete online *Instructor's Guide* (also downloadable) and supplementary teaching materials including a complete set of more than 700 *PowerPoint Slides* that may be used for lectures, and a test bank. Of course, all resources available for students (e.g, tiny tools, the Web References, the downloadable Reference Library) and professionals are also available.

The *Instructor's Guide for Software Engineering: A Practitioner's Approach* presents suggestions for conducting various types of software engineering courses, recommendations for a variety of software projects to be conducted in conjunction with a course, solutions to selected problems, and a number of useful teaching aids.

Professional Resources. A collection of resources available to industry practitioners (as well as students and faculty) includes outlines and samples of software engineering documents and other work products, a useful set of software engineering checklists,

a catalog of software engineering tools, a comprehensive collection of Web-based resources, and an "adaptable process model" that provides a detailed task breakdown of the software engineering process.

McGraw-Hill Connect® Computer Science provides online presentation, assignment, and assessment solutions. It connects your students with the tools and resources they'll need to achieve success. With Connect **Computer Science** you can deliver assignments, quizzes, and tests online. A robust set of questions and activities are presented and aligned with the textbook's learning outcomes. As an instructor, you can edit existing questions and author entirely new problems. Integrate grade reports easily with Learning Management Systems (LMS), such as WebCT and Blackboard—and much more. ConnectPlus® **Computer Science** provides students with all the advantages of Connect **Computer Science**, plus 24/7 online access to a media-rich eBook, allowing seamless integration of text, media, and assessments. To learn more, visit **www.mcgrawhillconnect.com**

McGraw-Hill LearnSmart® is available as a standalone product or an integrated feature of McGraw-Hill Connect **Computer Science**. It is an adaptive learning system designed to help students learn faster, study more efficiently, and retain more knowledge for greater success. LearnSmart assesses a student's knowledge of course content through a series of adaptive questions. It pinpoints concepts the student does not understand and maps out a personalized study plan for success. This innovative study tool also has features that allow instructors to see exactly what students have accomplished and a built-in assessment tool for graded assignments. Visit the following site for a demonstration. **www.mhlearnsmart.com**

Powered by the intelligent and adaptive LearnSmart engine, **SmartBook™** is the first and only continuously adaptive reading experience available today. Distinguishing what students know from what they don't, and honing in on concepts they are most likely to forget, SmartBook personalizes content for each student. Reading is no longer a passive and linear experience but an engaging and dynamic one, where students are more likely to master and retain important concepts, coming to class better prepared. SmartBook includes powerful reports that identify specific topics and learning objectives students need to study.

When coupled with its online support system, the eighth edition of *Software Engineering: A Practitioner's Approach,* provides flexibility and depth of content that cannot be achieved by a textbook alone.

With this edition of *Software Engineering: A Practitioner's Approach,* Bruce Maxim joins me (Roger Pressman) as a coauthor of the book. Bruce brought copious software engineering knowledge to the project and has added new content and insight that will be invaluable to readers of this edition.

Acknowledgments. Special thanks go to Tim Lethbridge of the University of Ottawa who assisted us in the development of UML and OCL examples, and developed the case study that accompanies this book, and Dale Skrien of Colby College, who developed the UML tutorial in Appendix 1. Their assistance and comments were invaluable.

In addition, we'd like to thank Austin Krauss, Senior Software Engineer at Treyarch, for providing insight into software development in the video game industry. We also wish to thank the reviewers of the eighth edition: Manuel E. Bermudez, University of Florida; Scott DeLoach, Kansas State University; Alex Liu, Michigan State University; and Dean Mathias, Utah State University. Their in-depth comments and thoughtful criticism have helped us make this a much better book.

Special Thanks. BRM: I am grateful to have had the opportunity to work with Roger on the eighth edition of this book. During the time I have been working on this book my son Benjamin shipped his first MobileApp and my daughter Katherine launched her interior design career. I am quite pleased to see the adults they have become. I am very grateful to my wife, Norma, for the enthusiastic support she has given me as I filled my free time with working on this book.

RSP: As the editions of this book have evolved, my sons, Mathew and Michael, have grown from boys to men. Their maturity, character, and success in the real world have been an inspiration to me. Nothing has filled me with more pride. They now have children of their own, Maya and Lily, who start still another generation. Both girls are already wizards on mobile computing devices. Finally, to my wife Barbara, my love and thanks for tolerating the many, many hours in the office and encouraging still another edition of "the book."

Roger S. Pressman
Bruce R. Maxim

CHAPTER

1

The Nature of Software

Key Concepts

application domains 6
cloud computing ... 10
failure curves 5
legacy software 8
mobile apps 10
product line..... 11
software, definition 4
software, questions about 4
software, nature of 3
wear 5
Webapps 9

As he finished showing me the latest build of one of the world's most popular first-person shooter video games, the young developer laughed.

"You're not a gamer, are you?" he asked.

I smiled. "How'd you guess?"

The young man was dressed in shorts and a tee shirt. His leg bounced up and down like a piston, burning the nervous energy that seemed to be commonplace among his co-workers.

"Because if you were," he said, "you'd be a lot more excited. You've gotten a peek at our next generation product and that's something that our customers would kill for . . . no pun intended."

We sat in a development area at one of the most successful game developers on the planet. Over the years, earlier generations of the game he demoed sold over 50 million copies and generated billions of dollars in revenue.

"So, when will this version be on the market?" I asked.

He shrugged. "In about five months, and we've still got a lot of work to do."

He had responsibility for game play and artificial intelligence functionality in an application that encompassed more than three million lines of code.

"Do you guys use any software engineering techniques?" I asked, half-expecting that he'd laugh and shake his head.

Quick Look

What is it? Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.

Who does it? Software engineers build and support software, and virtually everyone in the industrialized world uses it either directly or indirectly.

Why is it important? Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

What are the steps? Customers and other stakeholders express the need for computer software, engineers build the software product, and end users apply the software to solve a specific problem or to address a specific need.

What is the work product? A computer program that runs in one or more specific environments and services the needs of one or more end users.

How do I ensure that I've done it right? If you're a software engineer, apply the ideas contained in the remainder of this book. If you're an end user, be sure you understand your need and your environment and then select an application that best meets them both.

He paused and thought for a moment. Then he slowly nodded. "We adapt them to our needs, but sure, we use them."

"Where?" I asked, probing.

"Our problem is often translating the requirements the creatives give us."

"The creatives?" I interrupted.

"You know, the guys who design the story, the characters, all the stuff that make the game a hit. We have to take what they give us and produce a set of technical requirements that allow us to build the game."

"And after the requirements are established?"

He shrugged. "We have to extend and adapt the architecture of the previous version of the game and create a new product. We have to create code from the requirements, test the code with daily builds, and do lots of things that your book recommends."

"You know my book?" I was honestly surprised.

"Sure, used it in school. There's a lot there."

"I've talked to some of your buddies here, and they're more skeptical about the stuff in my book."

He frowned. "Look, we're not an IT department or an aerospace company, so we have to customize what you advocate. But the bottom line is the same—we need to produce a high-quality product, and the only way we can accomplish that in a repeatable fashion is to adapt our own subset of software engineering techniques."

"And how will your subset change as the years pass?"

He paused as if to ponder the future. "Games will become bigger and more complex, that's for sure. And our development timelines will shrink as more competition emerges. Slowly, the games themselves will force us to apply a bit more development discipline. If we don't, we're dead."

Quote:

"Ideas and technological discoveries are the driving engines of economic growth."

Wall Street Journal

Computer software continues to be the single most important technology on the world stage. And it's also a prime example of the law of unintended consequences. Sixty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the media); that software would be the driving force behind the personal computer revolution; that software applications would be purchased by consumers using their smart phones; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser; that a software company would become larger and more influential than all industrial-era companies; that a vast software-driven network would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.

No one could foresee that software would become embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial,

entertainment, office machines, . . . the list is almost endless. And if you believe the law of unintended consequences, there are many effects that we cannot yet predict.

No one could predict that millions of computer programs would have to be corrected, adapted, and enhanced as time passed. The burden of performing these "maintenance" activities would absorb more people and more resources than all work applied to the creation of new software.

As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect-oriented programming); and still others are broad-based (e.g., operating systems such as Linux). However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

This book presents a framework that can be used by those who build computer software—people who must get it right. The framework encompasses a process, a set of methods, and an array of tools that we call *software engineering*.

1.1 The Nature of Software

KEY POINT

Software is both a product and a vehicle that delivers a product.

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. Whether it resides within a mobile phone, a hand-held tablet, on the desktop, or within a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time—*information*. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms. It also provides a vehicle that can threaten personal privacy and a gateway that enables those with malicious intent to commit criminal acts.

Quote:

"Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets."

Brad J. Cox

The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build and protect complex systems.

Today, a huge software industry has become a dominant factor in the economies of the industrialized world. Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era. And yet, the questions that were asked of the lone programmer are the same questions that are asked when modern computer-based systems are built:[1]

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed—a concern that has led to the adoption of software engineering practice.

1.1.1 Defining Software

Today, most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:

? How should we define *software*?

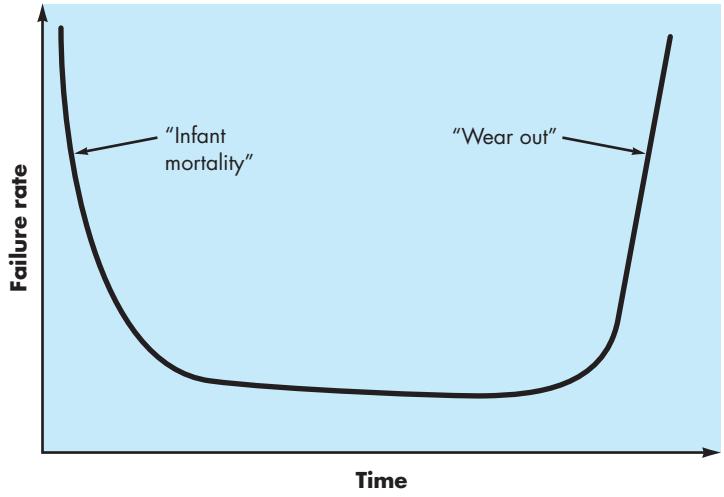
> Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered. But a more formal definition probably won't measurably improve your understanding.

1 In an excellent book of essays on the software business, Tom DeMarco [DeM95] argues the counterpoint. He states: "Instead of asking why software costs so much, we need to begin asking 'What have we done to make it possible for today's software to cost so little?' The answer to that question will help us continue the extraordinary level of achievement that has always distinguished the software industry."

FIGURE 1.1

Failure curve for hardware



ADVICE

If you want to reduce software deterioration, you'll have to do better software design (Chapters 12 to 18).

To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build. Software is a logical rather than a physical system element. Therefore, software has one fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*

Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out*.

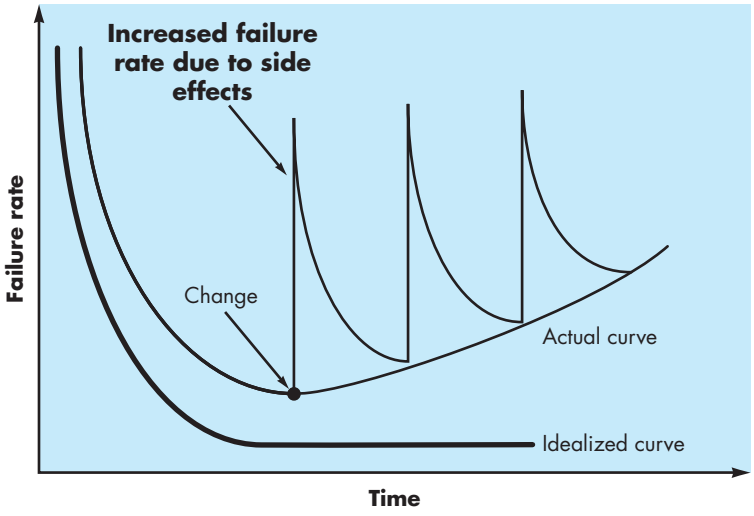
Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does *deteriorate!*

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life,[2] software will undergo change. As changes are

2 In fact, from the moment that development begins and long before the first version is delivered, changes may be requested by a variety of different stakeholders.

FIGURE 1.2

Failure curves for software



KEY POINT

Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve in Figure 1.2.

made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

1.1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

> **System software—**a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,[3] information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

3 Software is *determinate* if the order and timing of inputs, processing, and outputs is predictable. Software is *indeterminate* if the order and timing of inputs, processing, and outputs cannot be predicted in advance.

WebRef

One of the most comprehensive libraries of shareware/freeware can be found at shareware.cnet.com

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.

Engineering/scientific software—a broad array of "number-crunching programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer.

Web/Mobile applications—this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

Artificial intelligence software—makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Quote:

"What a computer is to me is the most remarkable tool that we have ever come up with. It's the equivalent of a bicycle for our minds."

Steve Jobs

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.

1.1.3 Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state-of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

> Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

? What do I do if I encounter a legacy system that exhibits poor quality?

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.[4] Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long. And yet, these systems support "core business functions and are indispensable to the business." What to do?

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:

? What types of changes are made to legacy systems?

- The software must be adapted to meet the needs of new computing environments or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with other more modern systems or databases.
- The software must be re-architected to make it viable within a evolving computing environment.

ADVICE

Every software engineer must recognize that change is natural. Don't try to fight it.

When these modes of evolution occur, a legacy system must be reengineered (Chapter 36) so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution;" that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other." [Day99]

4 In this case, quality is judged based on modern software engineering thinking—a somewhat unfair criterion since some modern software engineering concepts and principles may not have been well understood at the time that the legacy software was developed.

1.2 The Changing Nature of Software

Four broad categories of software are evolving to dominate the industry. And yet, these categories were in their infancy little more than a decade ago.

1.2.1 WebApps

In the early days of the World Wide Web (circa 1990 to 1995), *websites* consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications*[5] (we refer to these collectively as *WebApps*) were born.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

Quote:

"By the time we see any sort of stabilization, the Web will have turned into something completely different."

Louis Monier

A decade ago, WebApps "involve[d] a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology." [Pow98] But today, they provide full computing potential in many of the application categories noted in Section 1.1.2.

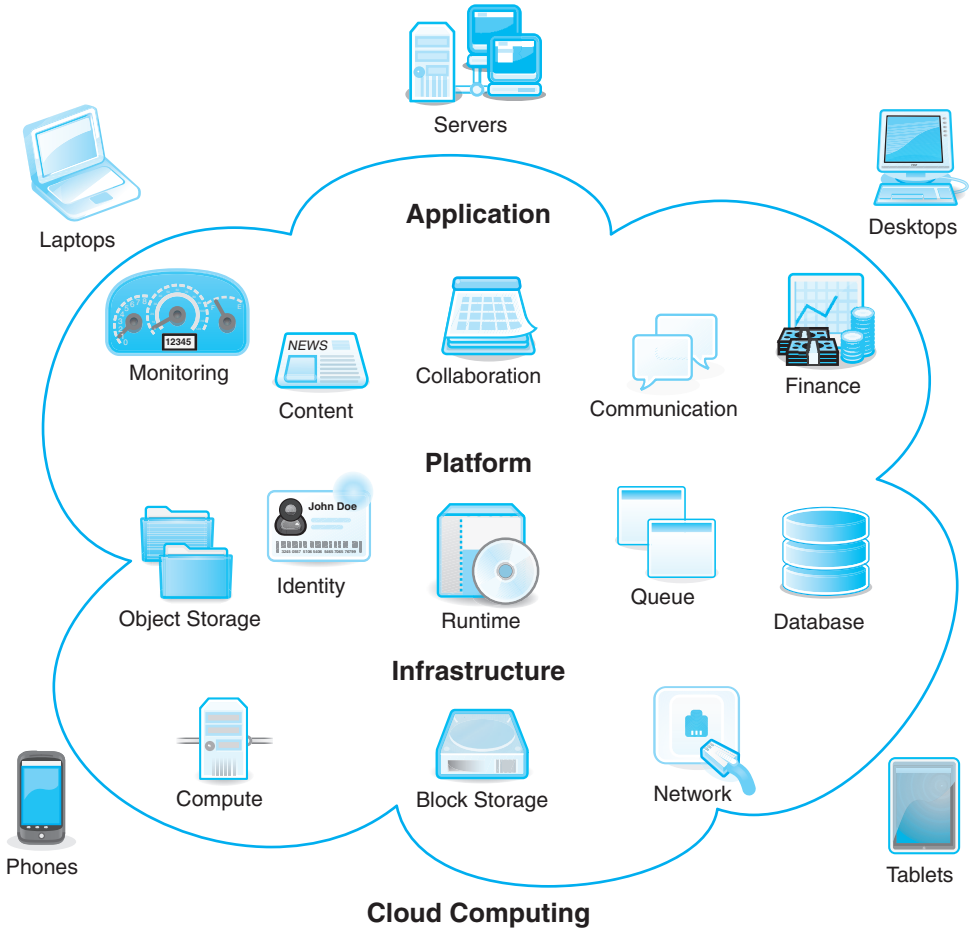
Over the past decade, Semantic Web technologies (often referred to as Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass "semantic databases [that] provide new functionality that requires Web linking, flexible [data] representation, and external access APIs." [Hen10] Sophisticated relational data structures will lead to entirely new WebApps that allow access to disparate information in ways never before possible.

1.2.2 Mobile Applications

The term *app* has evolved to connote software that has been specifically designed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile). In most instances, mobile applications encompass a user interface that takes advantage of the unique interaction mechanisms provided by the mobile platform, interoperability with Web-based resources that provide access to a wide array of information that is relevant to the app, and local processing capabilities that collect, analyze, and format information in a manner that is best suited to the mobile platform. In addition, a mobile app provides persistent storage capabilities within the platform.

5 In the context of this book, the term *Web application* (WebApp) encompasses everything from a simple Web page that might help a consumer compute an automobile lease payment to a comprehensive website that provides complete travel services for businesspeople and vacationers. Included within this category are complete websites, specialized functionality within websites, and information processing applications that reside on the Internet or on an intranet or extranet.

FIGURE 1.3 Cloud computing logical architecture [Wik13]



What is the difference between a WebApp and a mobile app?

It is important to recognize that there is a subtle distinction between mobile web applications and mobile apps. A *mobile web application* (WebApp) allows a mobile device to gain access to web-based content via a browser that has been specifically designed to accommodate the strengths and weaknesses of the mobile platform. A *mobile app* can gain direct access to the hardware characteristics of the device (e.g., accelerometer or GPS location) and then provide the local processing and storage capabilities noted earlier. As time passes, the distinction between mobile WebApps and mobile apps will blur as mobile browsers become more sophisticated and gain access to device level hardware and information.

1.2.3 Cloud Computing

Cloud computing encompasses an infrastructure or "ecosystem" that enables any user, anywhere, to use a computing device to share computing resources on a broad scale. The overall logical architecture of cloud computing is represented in Figure 1.3.

Referring to the figure, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure. In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data structures. In addition, devices can access executable applications that can be used in lieu of apps that reside on the computing device.

The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services. The *front-end* includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed. The *back-end* includes servers and related computing resources, data storage systems (e.g., databases), server-resident applications, and administrative servers that use middleware to coordinate and monitor traffic by establishing a set of protocols for access to the cloud and its resident resources. [Str08]

The cloud architecture can be segmented to provide access at a variety of different levels from full public access to private cloud architectures accessible only to those with authorization.

1.2.4 Product Line Software

The Software Engineering Institute defines a *software product line* as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [SEI13] The concept of a line of software products that are related in some way is not new. But the idea that a line of software products, all developed using the same underlying application and data architectures, and all implemented using a set of reusable software components that can be reused across the product line provides significant engineering leverage.

A software product line shares a set of assets that include requirements (Chapter 8), architecture (Chapter 13), design patterns (Chapter 16), reusable components (Chapter 14), test cases (Chapters 22 and 23), and other software engineering work products. In essence, a software product line results in the development of many products that are engineered by capitalizing on the commonality among all the products within the product line.

1.3 Summary

Software is the key element in the evolution of computer-based systems and products and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

Software—programs, data, and descriptive information—addresses a wide array of technology and application areas. Legacy software continues to present special challenges to those who must maintain it.

The nature of software is changing. Web-based systems and applications have evolved from simple collections of information content to sophisticated systems that present complex functionality and multimedia content. Although these WebApps have unique features and requirements, they are software nonetheless. Mobile applications present new challenges as apps migrate to a wide array of platforms. Cloud computing will transform the way in which software is delivered and the environment in which it exists. Product line software offers potential efficiencies in the manner in which software is built.

Problems and Points to Ponder

1.1. Provide at least five additional examples of how the law of unintended consequences applies to computer software.

1.2. Provide a number of examples (both positive and negative) that indicate the impact of software on our society.

1.3. Develop your own answers to the five questions asked at the beginning of Section 1.1. Discuss them with your fellow students.

1.4. Many modern applications change frequently—before they are presented to the end user and then after the first version has been put into use. Suggest a few ways to build software to stop deterioration due to change.

1.5. Consider the seven software categories presented in Section 1.1.2. Do you think that the same approach to software engineering can be applied for each? Explain your answer.

Further Readings and Information Sources[6]

Literally thousands of books are written about computer software. The vast majority discuss programming languages or software applications, but a few discuss software itself. Pressman and Herron (*Software Shock,* Dorset House, 1991) presented an early discussion (directed at the layperson) of software and the way professionals build it. Negroponte's best-selling book (*Being Digital*, Alfred A. Knopf, 1995) provides a view of computing and its overall impact in the twenty-first century. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) has produced a collection of amusing and insightful essays on software

6 The *Further Reading and Information Sources* section presented at the conclusion of each chapter presents a brief overview of print sources that can help to expand your understanding of the major topics presented in the chapter. We have created a comprehensive website to support *Software Engineering: A Practitioner's Approach* at **www.mhhe.com/pressman**. Among the many topics addressed within the website are chapter-by-chapter software engineering resources to Web-based information that can complement the material presented in each chapter. An Amazon.com link to every book noted in this section is contained within these resources.

and the process through which it is developed. Ray Kurzweil (*How to Create a Mind,* Viking, 2013) discusses how software will soon mimic human thought and lead to a "singularity" in the evolution of humans and machines.

Keeves (*Catching Digital,* Business Infomedia Online, 2012) discusses how business leaders must adapt as software evolves at an ever-increasing pace. Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do,* McGraw-Hill, 2000) argues that the "modern scourge" of software bugs can be eliminated and suggests ways to accomplish this. Books by Eubanks (*Digital Dead End: Fighting for Social Justice in the Information Age,* MIT Press, 2011) and Compaine (*Digital Divide: Facing a Crisis or Creating a Myth,* MIT Press, 2001) argue that the "divide" between those who have access to information resources (e.g., the Web) and those that do not is narrowing as we move into the first decade of this century. Books by Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design,* Morgan Kaufman, 2010), Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing,* New Riders Publishing, 2006), and Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications,* Auerbach, 2006) introduce the concept of "open-world" software and predict a wireless environment in which software must adapt to requirements that emerge in real time.

A wide variety of information sources that discuss the nature of software are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**

CHAPTER

2 Software Engineering

Key Concepts

framework activities 17
general principles . . 21
principles 21
problem solving . . . 19
SafeHome. 26
software engineering,
definition 15
layers 15
practice 19
software myths. 23
software process . . 16
umbrella activities. . 17

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application[1] has grown dramatically. *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*
- The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. *It follows that design becomes a pivotal activity.*

Quick Look

What is it? Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high-quality computer software.

Who does it? Software engineers apply the software engineering process.

Why is it important? Software engineering is important because it enables us to build complex systems in a timely manner and with high quality. It imposes discipline to work that can become quite chaotic, but it also allows the people who build computer software to adapt their approach in a manner that best suits their needs.

What are the steps? You build computer software like you build any successful product, by applying an agile, adaptable process that leads to a high-quality result that meets the needs of the people who will use the product. You apply a software engineering approach.

What is the work product? From the point of view of a software engineer, the work product is the set of programs, content (data), and other work products that are computer software. But from the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

How do I ensure that I've done it right? Read the remainder of this book, select those ideas that are applicable to the software that you build, and apply them to your work.

1 We will call these people "stakeholders" later in this book.

KEY POINT

Understand the problem before you build a solution.

- Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. *It follows that software should exhibit high quality.*
- As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow. *It follows that software should be maintainable.*

KEY POINT

Both quality and maintainability are an outgrowth of good design.

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book—*software engineering*.

2.1 Defining the Discipline

? How do we define software engineering?

The IEEE [IEE93a] has developed the following definition for software engineering:

> Software Engineering : (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies[2] foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

KEY POINT

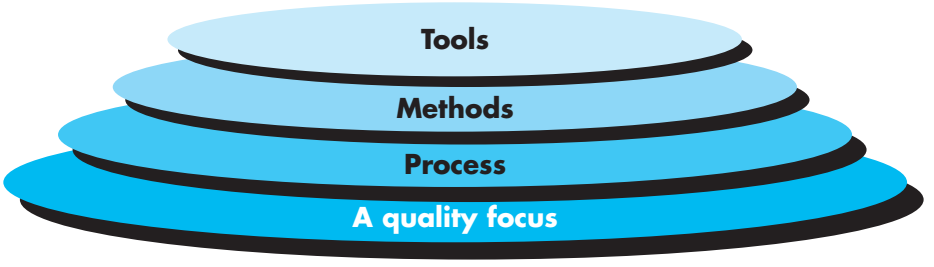
Software engineering encompasses a process, methods for managing and engineering software, and tools.

The foundation for software engineering is the *process* layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are

2 Quality management and related approaches are discussed throughout Part 3 of this book.

FIGURE 2.1

Software engineering layers



applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

WebRef

CrossTalk is a journal that provides pragmatic information on process, methods, and tools. It can be found at: www.stsc.hill.af.mil,

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

2.2 THE SOFTWARE PROCESS

? What are the elements of a software process?

A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created. An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural model). A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

Quote:

"A process defines who is doing *what* *when* and *how* to reach a certain goal."

Ivar Jacobson, Grady Booch, and James Rumbaugh

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

2.2.1 The Process Framework

? What are the five generic process framework activities?

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders).[3] The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Quote:

"Einstein argued that there must be a simplified explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity."

Fred Brooks

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of Web applications, and for the engineering

3 A *stakeholder* is anyone who has a stake in the successful outcome of the project—business managers, end users, software engineers, support people, etc. Rob Thomsett jokes that, "a stakeholder is a person holding a large and sharp stake . . . If you don't look after your stakeholders, you know where the stake will end up."

of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

2.2.2 Umbrella Activities

Software engineering process framework activities are complemented by a number of *umbrella activities*. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

KEY POINT
Umbrella activities occur throughout the software process and focus primarily on project management, tracking, and control.

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompass the activities required to create work products such as models, documents, logs, forms, and lists.

KEY POINT
Software process adaptation is essential for project success.

Each of these umbrella activities is discussed in detail later in this book.

2.2.3 Process Adaptation

Previously in this section, we noted that the software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team,

and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

- Overall flow of activities, actions, and tasks and the interdependencies among them.
- Degree to which actions and tasks are defined within each framework activity.
- Degree to which work products are identified and required.
- Manner in which quality assurance activities are applied.
- Manner in which project tracking and control activities are applied.
- Overall degree of detail and rigor with which the process is described.
- Degree to which the customer and other stakeholders are involved with the project.
- Level of autonomy given to the software team.
- Degree to which team organization and roles are prescribed.

In Part 1 of this book, we examine software process in considerable detail.

Quote:

"I feel a recipe is only a theme which an intelligent cook can play each time with a variation."

Madame Benoit

2.3 Software Engineering Practice

In Section 2.2, we introduced a generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work. But how does the practice of software engineering fit in? In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.[4]

WebRef

A variety of thought-provoking quotes on the practice of software engineering can be found at **www.literate-programming.com.**

2.3.1 The Essence of Practice

In the classic book, *How to Solve It,* written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

ADVICE

You might argue that Polya's approach is simply common sense. True. But it's amazing how often common sense is uncommon in the software world.

4 You should revisit relevant sections within this chapter as we discuss specific software engineering methods and umbrella activities later in this book.

In the context of software engineering, these commonsense steps lead to a series of essential questions [adapted from Pol45]:

ADVICE

The most important element of problem understanding is listening.

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing*. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the solution. Now you understand the problem (or so you think), and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Quote:

"There is a grain of discovery in the solution of any problem."

George Polya

Carry out the plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

2.3.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book we'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

ADVICE

Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:[5]

Quote:

"There is a certain majesty in simplicity which is far above all the quaintness of wit."

Alexander Pope (1688–1744)

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is no, don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it

5 Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at **http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment**.

often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume*

KEY POINT

If software has value, it will change over its useful life. For that reason, software must be built to be maintainable.

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.[6] This could very possibly lead to the reuse of an entire system.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.[7] Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code

6 This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

7 Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

The Seventh Principle: *Think!*

This last Principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

2.4 Software Development Myths

Software development myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

WebRef
The Software Project Managers Network at **www.spmn.com** can help you dispel these and other myths.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is no.

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

ADVICE

Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code

has been started), the cost impact is relatively small.[8] However, as time passes, the cost impact grows rapidly— resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by over 60 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

ADVICE

Whenever you think, we don't have time for software engineering, ask yourself, "Will we have time to do it over again?"

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review*. Software reviews (described in Chapter 20) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Today, most software professionals recognize the fallacy of the myths just described. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

8 Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 5.

2.5 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a "legacy system" to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

SafeHome[9]

How a Project Starts

The scene: Meeting room at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive vice president, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11n wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-businesspeople with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off the shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs and tablets have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer app. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as $30 to $40 million in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

With the exception of a passing reference, software was hardly mentioned as part of the conversation. And yet, software will make or break the *SafeHome* product line. The engineering effort will succeed only if *SafeHome* software succeeds.

9 The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are fictitious, but the situations and problems are real.

The market will accept the product only if the software embedded within it properly meets the customer's (as yet unstated) needs. We'll follow the progression of *SafeHome* software engineering in many of the chapters that follow.

2.6 Summary

Software engineering encompasses process, methods, and tools that enable complex computer-based systems to be built in a timely manner with quality. The software process incorporates five framework activities—communication, planning, modeling, construction, and deployment—that are applicable to all software projects. Software engineering practice is a problem-solving activity that follows a set of core principles.

A wide array of software myths continue to lead managers and practitioners astray, even as our collective knowledge of software and the technologies required to build it grows. As you learn more about software engineering, you'll begin to understand why these myths should be debunked whenever they are encountered.

Problems and Points to Ponder

2.1. Figure 2.1 places the three software engineering layers on top of a layer entitled "A quality focus." This implies an organizational quality program such as total quality management. Do a bit of research and develop an outline of the key tenets of a total quality management program.

2.2. Is software engineering applicable when WebApps are built? If so, how might it be modified to accommodate the unique characteristics of WebApps?

2.3. As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a doomsday but realistic scenario in which the failure of a computer program could do great harm, either economic or human.

2.4. Describe a process framework in your own words. When we say that framework activities are applicable to all projects, does this mean that the same work tasks are applied for all projects, regardless of size and complexity? Explain.

2.5. Umbrella activities occur throughout the software process. Do you think they are applied evenly across the process, or are some concentrated in one or more framework activities?

2.6. Add two additional myths to the list presented in Section 2.4. Also state the reality that accompanies the myth.

Further Readings and Information Sources

The current state of the software engineering and the software process can best be determined from publications such as *IEEE Software, IEEE Computer, CrossTalk,* and *IEEE Transactions on Software Engineering*. Industry periodicals such as *Application Development Trends* and *Cutter IT Journal* often contain articles on software engineering topics. The discipline is "summarized" every year in the *Proceeding of the International Conference*

on Software Engineering, sponsored by the IEEE and ACM, and is discussed in depth in journals such as *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes,* and *Annals of Software Engineering*. Tens of thousands of Web pages are dedicated to software engineering and the software process.

Many books addressing the software process and software engineering have been published in recent years. Some present an overview of the entire process, while others delve into a few important topics to the exclusion of others. Among the more popular offerings (in addition to this book!) are

SWEBOK: Guide to the Software Engineering Body of Knowledge,[10] IEEE, 2013, see: http://www.computer.org/portal/web/swebok

Andersson, E., et al., *Software Engineering for Internet Applications,* MIT Press, 2006.

Braude, E., and M. Bernstein, *Software Engineering: Modern Approaches,* 2nd ed., Wiley, 2010.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Hussain, S., *Software Engineering,* I K International Publishing House, 2013.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2nd ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* 3rd ed., Springer, 2010.

Pfleeger, S., *Software Engineering: Theory and Practice,* 4th ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering,* 8th ed., McGraw-Hill, 2010.

Sommerville, I., *Software Engineering,* 9th ed., Addison-Wesley, 2010.

Stober, T., and U. Hansmann, *Agile Software Development: Best Practices for Large Development Projects,* Springer, 2009.

Tsui, F., and O. Karam, *Essentials of Software Engineering,* 2nd ed., Jones & Bartlett Publishers, 2009.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007), Richardson and Gwaltney (*Ship it! A Practical Guide to Successful Software Projects,* Pragmatic Bookshelf, 2005), and Humble and Farley (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010) present a broad collection of useful guidelines that are applicable to the deployment activity.

Many software engineering standards have been published by the IEEE, ISO, and their standards organizations over the past few decades. Moore (*The Road Map to Software Engineering: A Standards-Based Guide,* IEEE Computer Society Press [Wiley], 2006) provides a useful survey of relevant standards and how they apply to real projects.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**

10 Available free of charge at **<http://www.computer.org/portal/web/swebok/htmlformat>**

PART

One

The Software Process

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the process that provides a framework for software engineering practice. These questions are addressed in the chapters that follow:

- What is a software process?
- What are the generic framework activities that are present in every software process?
- How are processes modeled and what are process patterns?
- What are the prescriptive process models and what are their strengths and weaknesses?
- Why is *agility* a watchword in modern software engineering work?
- What is agile software development and how does it differ from more traditional process models?

Once these questions are answered you'll be better prepared to understand the context in which software engineering practice is applied.

CHAPTER

3 Software Process Structure

Key Concepts

generic process model..... 31
process assessment..... 37
process flow..... 31
process improvement 38
process patterns..... 35
task set 34

In a fascinating book that provides an economist's view of software and software engineering, Howard Baetjer Jr. [Bae98] comments on the software process:

> Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become the software is brought together and embodied in the software. The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]. It is an iterative process in which the evolving tool itself serves as the medium for communication, with each new round of the dialogue eliciting more useful knowledge from the people involved.

Indeed, building computer software is an iterative social learning process, and the outcome, something that Baetjer would call "software capital," is an embodiment of knowledge collected, distilled, and organized as the process is conducted.

Quick Look

What is it? When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a "software process."

Who does it? Software engineers and their managers adapt the process to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

Why is it important? Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? At a detailed level, the process that you adopt depends on the software that you're building. One process might be appropriate for creating software for an aircraft avionics system, while an entirely different process would be indicated for the creation of a website.

What is the work product? From the point of view of a software engineer, the work products are the programs, documents, and data that are produced as a consequence of the activities and tasks defined by the process.

How do I ensure that I've done it right? There are a number of software process assessment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.

But what exactly is a software process from a technical point of view? Within the context of this book, we define a *software process* as a framework for the activities, actions, and tasks that are required to build high-quality software. Is "process" synonymous with "software engineering"? The answer is yes and no. A software process defines the approach that is taken as software is engineered. But software engineering also encompasses technologies that populate the process—technical methods and automated tools.

More important, software engineering is performed by creative, knowledgeable people who should adapt a mature software process so that it is appropriate for the products that they build and the demands of their marketplace.

3.1 A Generic Process Model

In Chapter 2, a process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another.

KEY POINT

The hierarchy of technical work within the software process is activities, encompassing actions, populated by tasks.

The software process is represented schematically in Figure 3.1. Referring to the figure, each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

As we discussed in Chapter 2, a generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment**. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

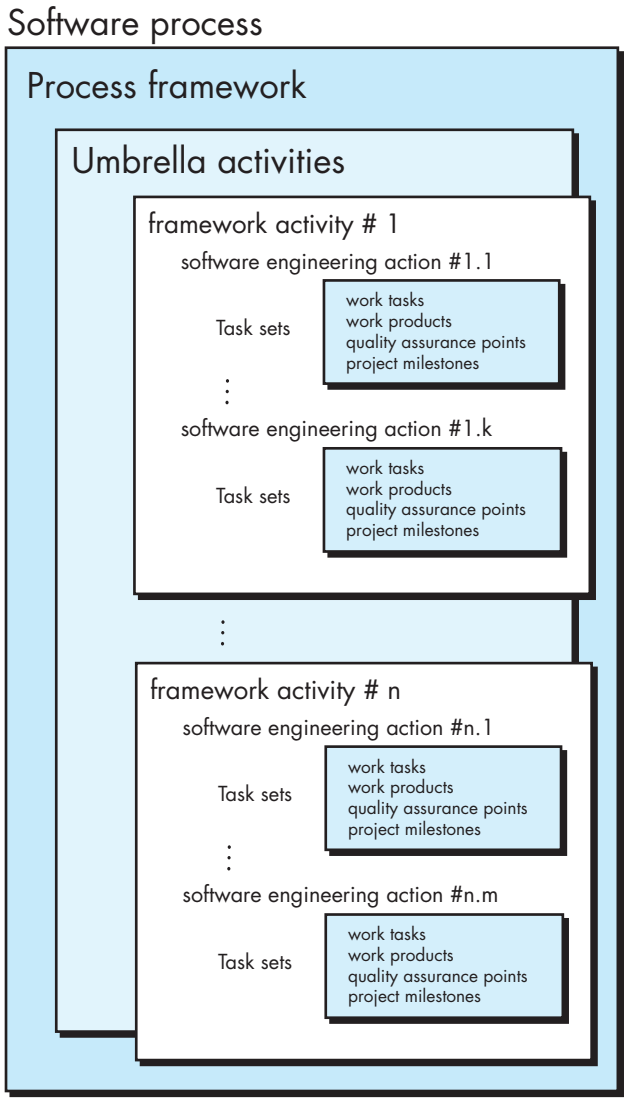
? What is process flow?

You should note that one important aspect of the software process has not yet been discussed. This aspect—called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 3.2.

A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 3.2a). An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 3.2b). An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 3.2c). A *parallel process flow* (Figure 3.2d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

FIGURE 3.1

A software process framework



3.2 DEFINING A FRAMEWORK ACTIVITY

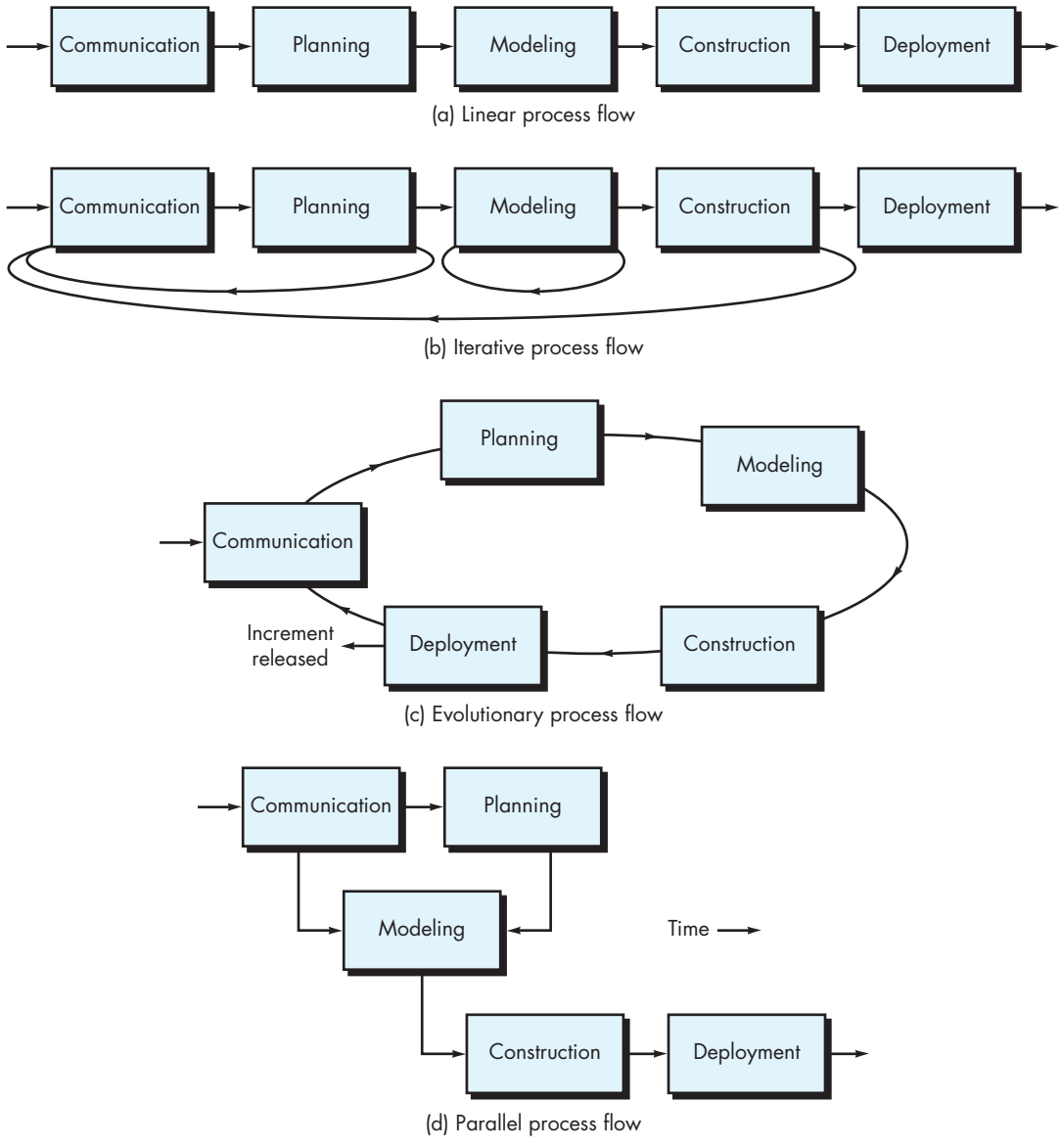
Quote:

"If the process is right, the results will take care of themselves."

Takashi Osada

Although we have described five framework activities and provided a basic definition of each in Chapter 2, a software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

FIGURE 3.2 Process flow



? How does a framework activity change as the nature of the project changes?

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the **communication** activity might encompass little more than a phone call or email with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.
2. Discuss requirements and develop notes.

3. Organize notes into a brief written statement of requirements.
4. Email to stakeholder for review and approval.

KEY POINT

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions (described in Chapter 8): *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

3.3 Identifying a Task Set

Referring again to Figure 3.1, each software engineering action (e.g., *elicitation,* an action associated with the **communication** activity) can be represented by a number of different *task sets*—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

INFO

Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the **communication** activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

For a larger, more complex software project, a different task set would be required. It might encompass the following work tasks:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Both of these task sets achieve "requirements gathering," but they are quite different in their depth and formality. The software team chooses the task set that will allow it to achieve the goal of each action and still maintain quality and agility.

You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

3.4 Process Patterns

What is a process pattern?

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern*[1] describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template [Amb98]—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Quote:

"The repetition of patterns is quite a different thing than the repetition of parts. Indeed, the different parts will be unique because the patterns are the same."

Christopher Alexander

Patterns can be defined at any level of abstraction.[2] In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler [Amb98] has proposed a template for describing a process pattern:

KEY POINT

A pattern template provides a consistent means for describing a pattern.

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specified. Ambler [Amb98] suggests three types:

1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication**. This pattern would incorporate the task pattern **RequirementsGathering** and others.

1 A detailed discussion of patterns is presented in Chapter 11

2 Patterns are applicable to many software engineering activities. Analysis, design, and testing patterns are discussed in Chapters 11, 13, 15, 16, and 20. Patterns and "antipatterns" for project management activities are discussed in Part 4 of this book.

2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).
3. ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping**.[3]

Quote:

"We think that software developers are missing a vital truth: most organizations don't know what they do. They think they know, but they don't know."

Tom DeMarco

Initial Context. Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists?

For example, the **Planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns [specified] for the **Communication** pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: **ProjectTeam, CollaborativeGuidelines, ScopeIsolation**, **RequirementsGathering, ConstraintDescription,** and **ScenarioCreation.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the **Deployment** activity is under way.

3 These phase patterns are discussed in Chapter 4.

WebRef

Comprehensive resources on process patterns can be found at www.ambysoft.com/processPatternsPage.html.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern). The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

INFO

An Example Process Pattern

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements.

Pattern Name. **RequirementsUnclear**

Intent. This pattern describes an approach for building a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.

Type. Phase pattern.

Initial Context. The following conditions must be met prior to the initiation of this pattern: (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders; (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem. Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem to be solved, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solution. A description of the prototyping process would be presented here and is described later in Section 4.1.3.

Resulting Context. A software prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

Related Patterns. The following patterns are related to this pattern: **CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.**

Known Uses and Examples. Prototyping is recommended when requirements are uncertain.

3.5 Process Assessment and Improvement

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapter 19). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself can be assessed to

KEY POINT

Assessment attempts to understand the current state of the software process with the intent of improving it.

ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.[4]

A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement (SCAMPI)— provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].

Quote:

"Software organizations have exhibited significant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

CMM-Based Appraisal for Internal Process Improvement (CBA IPI)— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

A more detailed discussion of software assessment and process improvement methods is presented in Chapter 37.

3.6 Summary

A generic process model for software engineering encompasses a set of framework and umbrella activities, actions, and work tasks. Each of a variety of process models can be described by a different process flow—a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

Problems and Points to Ponder

3.1. In the introduction to this chapter Baetjer notes: "The process provides interaction between users and designers, between users and evolving tools, and between designers and evolving tools [technology]." List five questions that (1) designers should ask users, (2) users should ask designers, (3) users should ask themselves about the software product that is to be built, (4) designers should ask themselves about the software product that is to be built and the process that will be used to build it.

4 The SEI's CMMI [CMM07] describes the characteristics of a software process and the criteria for a successful process in voluminous detail.

3.2. Discuss the differences among the various process flows described in Section 3.1. Can you identify types of problems that might be applicable to each of the generic flows described?

3.3. Try to develop a set of actions for the communication activity. Select one action and define a task set for it.

3.4. A common problem during **communication** occurs when you encounter two stakeholders who have conflicting ideas about what the software should be. That is, you have mutually conflicting requirements. Develop a process pattern (this would be a stage pattern) using the template presented in Section 3.4 that addresses this problem and suggest an effective approach to it.

Further Readings and Information Sources

Most software engineering textbooks consider process models in some detail. Books by Sommerville (*Software Engineering,* 9th ed., Addison-Wesley, 2010), Schach (*Object-Oriented and Classical Software Engineering,* 8th ed., McGraw-Hill, 2010) and Pfleeger and Atlee (*Software Engineering: Theory and Practice,* 4th ed., Prentice Hall, 2009) consider traditional paradigms and discuss their strengths and weaknesses. Munch and his colleagues (*Software Process Definition and Management,* Springer, 2012) present a software and systems engineering view of the process and the product. Glass (*Facts and Fallacies of Software Engineering,* Prentice Hall, 2002) provides an unvarnished, pragmatic view of the software engineering process. Although not specifically dedicated to process, Brooks (*The Mythical Man-Month,* 2nd ed., Addison-Wesley, 1995) presents age-old project wisdom that has everything to do with process.

Firesmith and Henderson-Sellers (*The OPEN Process Framework: An Introduction,* Addison-Wesley, 2001) present a general template for creating "flexible, yet discipline software processes" and discuss process attributes and objectives. Madachy (*Software Process Dynamics,* Wiley-IEEE, 2008) discusses modeling techniques that allow the interrelated technical and social elements of the software process to be analyzed. Sharpe and McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development,* 2nd ed., Artech House, 2008) present tools for modeling both software and business processes.

A wide variety of information sources on software engineering and the software process are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**

CHAPTER

4 Process Models

Key Concepts

aspect-oriented software development..... 54
component-based development..... 53
concurrent models... 49
evolutionary process model..... 45
formal methods model..... 53
incremental process models..... 43
Personal Software Process 59
process modeling tools 62
process technology . 61
prototyping 45
spiral model 47
Team Software Process 60
unified process 55
V-model..... 42
waterfall model ... 41

Process models were originally proposed to bring order to the chaos of software development. History has indicated that these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the products that are produced remain on "the edge of chaos."

In an intriguing paper on the strange relationship between order and chaos in the software world, Nogueira and his colleagues [Nog00] state

> The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise." [Kau95] The edge of chaos can be visualized as an unstable, partially structured state . . . It is unstable because it is constantly attracted to chaos or to absolute order.
>
> We have the tendency to think that order is the ideal state of nature. This could be a mistake. Research . . . supports the theory that operation away from equilibrium generates creativity, self-organized processes, and increasing returns [Roo96]. Absolute order means the absence of variability, which could be an

Quick Look

What is it? A process model provides a specific roadmap for software engineering work. It defines the flow of all activities, actions and tasks, the degree of iteration, the work products, and the organization of the work that must be done.

Who does it? Software engineers and their managers adapt a process model to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it.

Why is it important? Because process provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? The process model provides you with the "steps" you'll need to perform disciplined software engineering work.

What is the work product? From the point of view of a software engineer, the work product is a customized description of the activities and tasks defined by the process.

How do I ensure that I've done it right? There are a number of software process assessment mechanisms that enable organizations to determine the "maturity" of their software process. However, the quality, timeliness, and long-term viability of the product you build are the best indicators of the efficacy of the process that you use.

advantage under unpredictable environments. Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder.

KEY POINT

The purpose of process models is to try to reduce the chaos present in developing new software products.

The philosophical implications of this argument are significant for software engineering. Each process model described in this chapter tries to strike a balance between the need to impart order in a chaotic world and the need to be adaptable when things change constantly.

4.1 Prescriptive Process Models

WebRef

An award-winning "process simulation game" that includes most important prescriptive process models can be found at: **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html**.

A *prescriptive process model*[1] strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. But are prescriptive models appropriate for a software world that thrives on change? If we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, we examine the prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

All software process models can accommodate the generic framework activities described in Chapters 2 and 3, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

4.1.1 The Waterfall Model

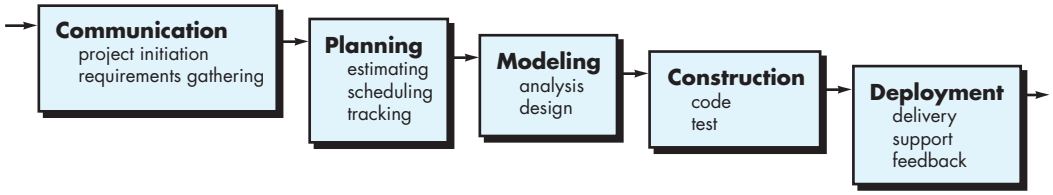
KEY POINT

Prescriptive process models define a prescribed set of process elements and a predictable process work flow.

There are times when the requirements for a problem are well understood—when work flows from communication through deployment in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

1 Prescriptive process models are sometimes referred to as "traditional" process models.

FIGURE 4.1 The waterfall model



KEY POINT
The V-model illustrates how verification and validation actions are associated with earlier engineering actions.

The *waterfall model,* sometimes called the *classic life cycle*, suggests a systematic, sequential approach[2] to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 4.1).

A variation in the representation of the waterfall model is called the *V-model*. Represented in Figure 4.2, the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side.[3] In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. However, over the past four decades, criticism of this process model has caused even ardent supporters to question its efficacy [Han95]. Among the problems that are sometimes encountered when the waterfall model is applied are:

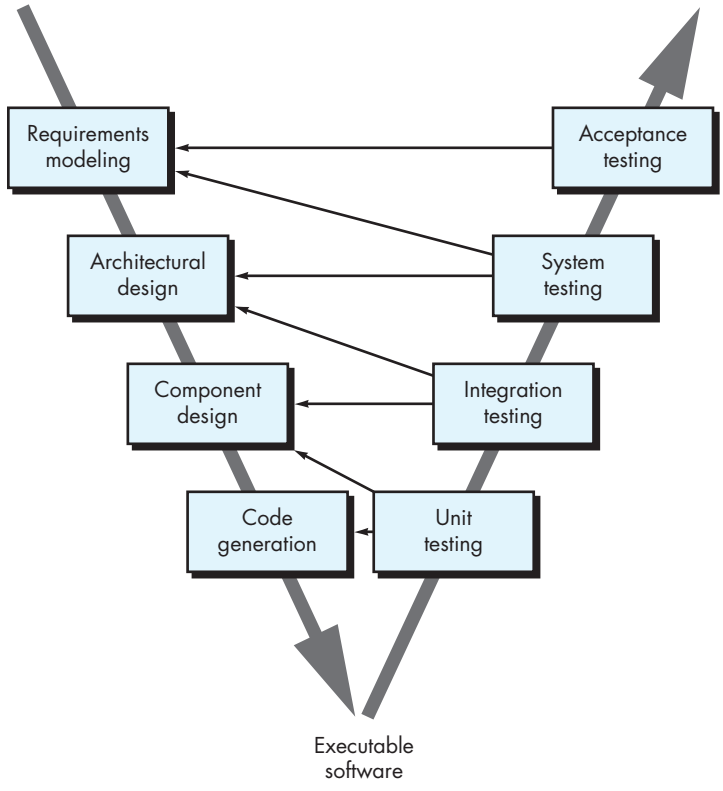
? Why does the waterfall model sometimes fail?

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

2 Although the original waterfall model proposed by Winston Royce [Roy70] made provision for "feedback loops," the vast majority of organizations that apply this process model treat it as if it were strictly linear.

3 A detailed discussion of quality assurance actions is presented in Part 3 of this book.

FIGURE 4.2
The V-model



3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Quote:

"Too often, software work follows the first law of bicycling: No matter where you're going, it's uphill and against the wind."

Author unknown

In an interesting analysis of actual projects, Bradac [Bra94] found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process.

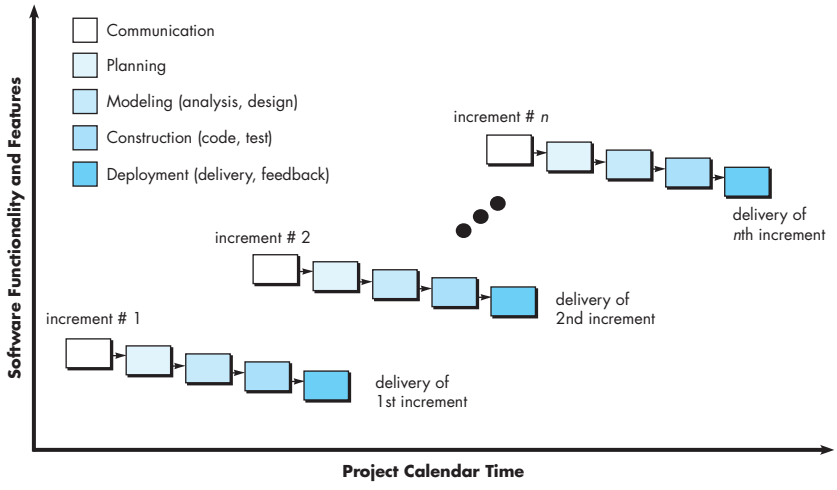
Today, software work is fast paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

4.1.2 Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely

FIGURE 4.3

The incremental model



linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

KEY POINT

The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

The incremental model combines the elements' linear and parallel process flows discussed in Chapter 3. Referring to Figure 4.3, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93].

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm discussed in the next subsection.

ADVICE

Your customer demands delivery by a date that is impossible to meet. Suggest delivering one or more increments by that date and the rest of the software (additional increments) later.

When an incremental model is used, the first increment is often a *core product*. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

4.1.3 Evolutionary Process Models

KEY POINT

Evolutionary process models produce an increasingly more complete version of the software with each iteration.

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that grows and changes.

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. In the paragraphs that follow, we present two common evolutionary process models.

Quote:

"Plan to throw one away. You will do that, anyway. Your only choice is whether to try to sell the throwaway to customers."

Frederick P. Brooks

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

ADVICE

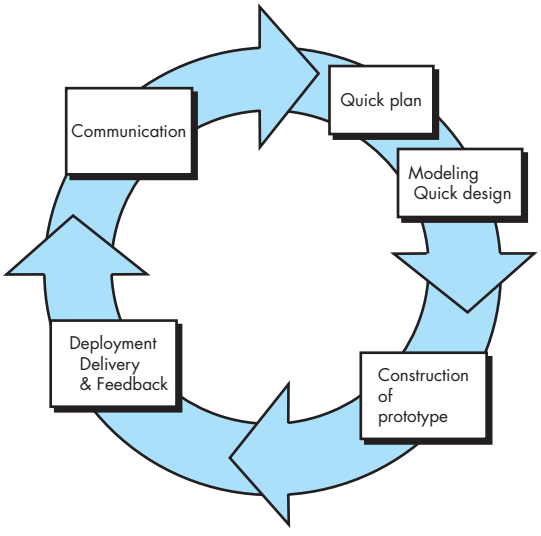
When your customer has a legitimate need, but is clueless about the details, develop a prototype as a first step.

The prototyping paradigm (Figure 4.4) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats). The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

FIGURE 4.4

The prototyping paradigm



But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] provides one answer:

> In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

ADVICE

Resist pressure to extend a rough prototype into a production product. Quality almost always suffers as a result.

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is

SafeHome

Selecting a Process Model, Part 1

The scene: Meeting room for the software engineering group at CPI Corporation, a (fictional) company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member; and Ed Robbins, software team member.

The conversation:

Lee: So let's recapitulate. I've spent some time discussing the *SafeHome* product line as we see it at the moment. No doubt, we've got a lot of work to do to simply define the thing, but I'd like you guys to begin thinking about how you're going to approach the software part of this project.

Doug: Seems like we've been pretty disorganized in our approach to software in the past.

Ed: I don't know, Doug, we always got product out the door.

Doug: True, but not without a lot of grief, and this project looks like it's bigger and more complex than anything we've done in the past.

Jamie: Doesn't look that hard, but I agree . . . our ad hoc approach to past projects won't work here, particularly if we have a very tight time line.

Doug (smiling): I want to be a bit more professional in our approach. I went to a short course last week and learned a lot about software engineering . . . good stuff. We need a process here.

Jamie (with a frown): My job is to build computer programs, not push paper around.

Doug: Give it a chance before you go negative on me. Here's what I mean. (Doug proceeds to describe the process framework described in Chapter 3 and the prescriptive process models presented to this point.)

Doug: So anyway, it seems to me that a linear model is not for us . . . assumes we have all requirements up front and, knowing this place, that's not likely.

Vinod: Yeah, and it sounds way too IT-oriented . . . probably good for building an inventory control system or something, but it's just not right for *SafeHome*.

Doug: I agree.

Ed: That prototyping approach seems okay. A lot like what we do here anyway.

Vinod: That's a problem. I'm worried that it doesn't provide us with enough structure.

Doug: Not to worry. We've got plenty of other options, and I want you guys to pick what's best for the team and best for the project.

available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

The Spiral Model. Originally proposed by Barry Boehm [Boe88], the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more

complete versions of the software. Boehm [Boe01a] describes the model in the following manner:

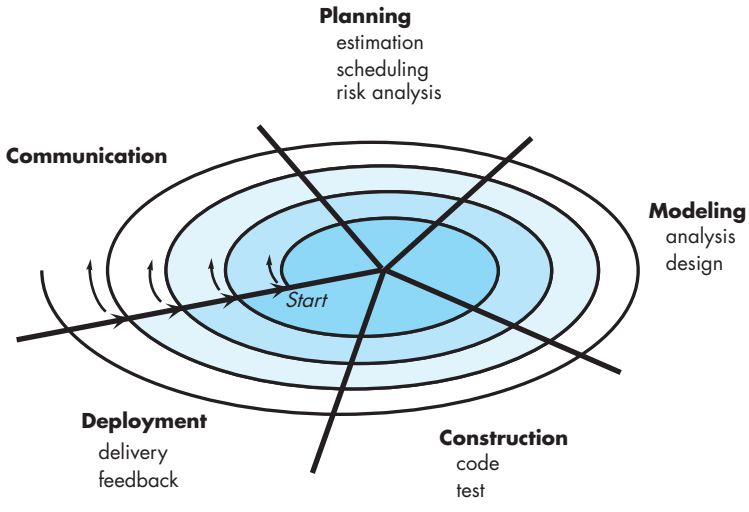
> The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

KEY POINT
The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, we use the generic framework activities discussed earlier.[4] Each of the framework activities represent one segment of the spiral path illustrated in Figure 4.5. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk (Chapter 35) is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

FIGURE 4.5
A typical spiral model



4 The spiral model discussed in this section is a variation on the model proposed by Boehm. For further information on the original spiral model, see [Boe88]. More recent discussion of Boehm's spiral model can be found in [Boe98].

WebRef

Useful information about the spiral model can be obtained at: **www.sei.cmu.edu/publications/documents/00.reports/00sr008.html**.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations[5] until concept development is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

ADVICE

If your management demands fixed-budget development (generally a bad idea), the spiral can be a problem. As each circuit is completed, project cost is revisited and revised.

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

Quote:

"I'm only this far and only tomorrow leads my way."

Dave Matthews Band

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

4.1.4 Concurrent Models

The *concurrent development model,* sometimes called *concurrent engineering,* allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter. For example, the modeling activity

5 The arrows pointing inward along the axis separating the *deployment* region from the *communication* region indicate a potential for local iteration along the same spiral path.

SafeHome

Selecting a Process Model, Part 2

The scene: Meeting room for the software engineering group at CPI Corporation, a company that makes consumer products for home and commercial use.

The players: Lee Warren, engineering manager; Doug Miller, software engineering manager; Vinod and Jamie, members of the software engineering team.

The conversation: (Doug describes evolutionary process options.)

Jamie: Now I see something I like. An incremental approach makes sense, and I really like the flow of that spiral model thing. That's keepin' it real.

Vinod: I agree. We deliver an increment, learn from customer feedback, re-plan, and then deliver another increment. It also fits into the nature of the product. We can have something on the market fast and then add functionality with each version, er, increment.

Lee: Wait a minute. Did you say that we regenerate the plan with each tour around the spiral, Doug? That's not so great; we need one plan, one schedule, and we've got to stick to it.

Doug: That's old-school thinking, Lee. Like the guys said, we've got to keep it real. I submit that it's better to tweak the plan as we learn more and as changes are requested. It's way more realistic. What's the point of a plan if it doesn't reflect reality?

Lee (frowning): I suppose so, but . . . senior management's not going to like this . . . they want a fixed plan.

Doug (smiling): Then you'll have to reeducate them, buddy.

KEY POINT

Project plans must be viewed as living documents; progress must be assessed often and revised to take changes into account.

ADVICE

The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

Quote:

"Every process in your organization has a customer, and without a customer a process has no purpose."

V. Daniel Hunt

defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.[6]

Figure 4.6 provides an example of the concurrent modeling approach. An activity—**modeling**—may be in any one of the states[7] noted at any given time. Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **none** state while initial communication was completed) now makes a transition into the **under development** state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

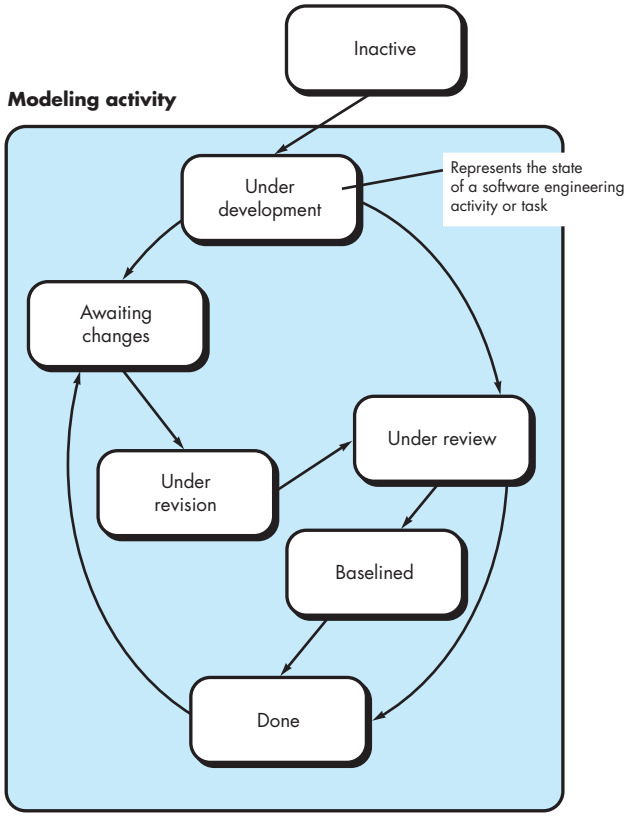
Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements

6 It should be noted that analysis and design are complex tasks that require substantial discussion. Part 2 of this book considers these topics in detail.

7 A *state* is some externally observable mode of behavior.

FIGURE 4.6

One element of the concurrent process model



model is uncovered. This generates the event *analysis model correction*, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network. Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states associated with each activity.

4.1.5 A Final Word on Evolutionary Processes

We have already noted that modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction. In many cases, time-to-market is the most important management

requirement. If a market window is missed, the software project itself may be meaningless.[8]

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. These are summarized by Nogueira and his colleagues [Nog00]:

> Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product . . .
>
> Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected . . .
>
> Third, [evolutionary] software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary.

? What are the potential weaknesses of evolutionary process models?

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts (e.g., [You95], [Bac97]).

The intent of evolutionary models is to develop high-quality software[9] in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

4.2 Specialized Process Models

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.[10]

8 It is important to note, however, that being the first to reach a market is no guarantee of success. In fact, many very successful software products have been second or even third to reach the market (learning from the mistakes of their predecessors).

9 In this context software quality is defined quite broadly to encompass not only customer satisfaction, but also a variety of technical criteria discussed in Part 2 of this book.

10 In some cases, these specialized process models might better be characterized as a collection of techniques or a "methodology" for accomplishing a specific software development goal. However, they do imply a process.

4.2.1 Component-Based Development

WebRef
Useful information on component-based development can be obtained at: **www.cbd-hq.com**.

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component-based development model comprises applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages[11] of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits including a reduction in development cycle time and a reduction in project cost if component reuse becomes part of your organization's culture. Component-based development is discussed in more detail in Chapter 14.

4.2.2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called *cleanroom software engineering* [Mil87, Dye92], is currently applied by some software development organizations.

11 Object-oriented concepts are discussed in Appendix 2 and are used throughout Part 2 of this book. In this context, a class encompasses a set of data and the procedures that process the data. A package of classes is a collection of related classes that work together to achieve some end result.

When formal methods (Appendix 3) are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

? If formal methods can demonstrate software correctness, why is it they are not widely used?

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

4.2.3 Aspect-Oriented Software Development

WebRef
A wide array of resources and information on AOP can be found at: **aosd.net**.

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object-oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

KEY POINT
AOSD defines "aspects" that express customer concerns that cut across multiple system functions, features, and information.

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*. *Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture. *Aspect-oriented software development* (AOSD), often referred to as *aspect-oriented programming* (AOP) or *aspect-oriented component engineering* (AOCE) [Gru02], is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and

constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern" [Elr01].

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

A detailed discussion of aspect-oriented software development is best left to books dedicated to the subject. If you have further interest, see [Ras11], [Saf08], [Cla05], [Fil05], [Jac04], and [Gra03].

SOFTWARE TOOLS

Process Management

Objective: To assist in the definition, execution, and management of prescriptive process models.

Mechanics: Process management tools allow a software organization or team to define a complete software process model (framework activities, actions, tasks, QA checkpoints, milestones, and work products). In addition, the tools provide a road map as software engineers do technical work and a template for managers who must track and control the software process.

Representative tools:[12]

GDPA, a research process definition tool suite, developed at Bremen University in Germany (**www.informatik.uni-bremen.de/uniform/gdpa/home.htm**), provides a wide array of process modeling and management functions.

ALM Studio, developed by Kovair Corporation (**http://www.kovair.com/**) encompasses a suite of tools for process definition, requirements management, issue resolution, project planning, and tracking.

ProVision BPMx, developed by OpenText (**http://bps.opentext.com/**), is representative of many tools that assist in process definition and workflow automation.

A worthwhile listing of many different tools associated with the software process can be found at **www.computer.org/portal/web/swebok/html/ch10**.

4.3 THE UNIFIED PROCESS

In their seminal book on the *Unified Process (UP),* Ivar Jacobson, Grady Booch, and James Rumbaugh [Jac99] discuss the need for a "use case driven, architecture-centric, iterative and incremental" software process when they state:

> Today, the trend in software is toward bigger, more complex systems. That is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of

12 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

> the Internet for exchanging all kinds of information . . . Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

In some ways the Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of agile software development (Chapter 5). The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system (the use case).[13] It emphasizes the important role of software architecture and "helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse" [Jac99]. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

4.3.1 A Brief History

During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method" that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts (e.g., [Wir90]) in object-oriented modeling. The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems. By 1997, UML became a de facto industry standard for object-oriented software development.

UML is used throughout Part 2 of this book to represent both requirements and design models. Appendix 1 presents an introductory tutorial for those who are unfamiliar with basic UML notation and modeling rules. A comprehensive presentation of UML is best left to textbooks dedicated to the subject. Recommended books are listed in Appendix 1.

4.3.2 Phases of the Unified Process[14]

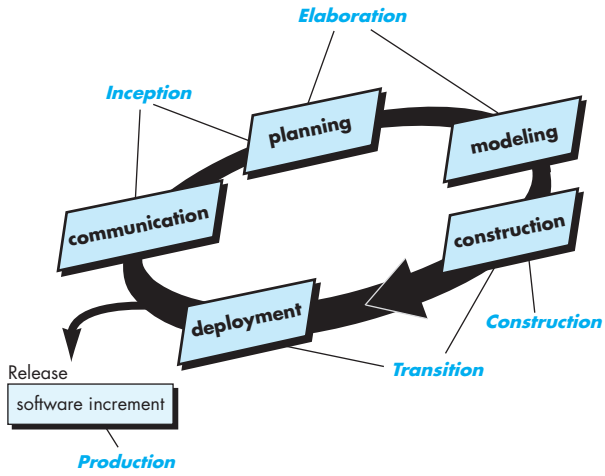
In Chapter 3, we discussed five generic framework activities and argued that they may be used to describe any software process model. The Unified Process

13 A *use case* (Chapter 8) is a text narrative or template that describes a system function or feature from the user's point of view. A use case is written by the user and serves as a basis for the creation of a more comprehensive analysis model.

14 The Unified Process is sometimes called the *Rational Unified Process* (RUP) after the Rational Corporation (subsequently acquired by IBM), an early contributor to the development and refinement of the UP and a builder of complete environments (tools and technology) that support the process.

FIGURE 4.7

The Unified Process



is no exception. Figure 4.7 depicts the "phases" of the UP and relates them to the generic activities that have been discussed in Chapter 1 and earlier in this chapter.

KEY POINT

UP *phases* are similar in intent to the generic framework activities defined in this book.

The *inception phase* of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases (Chapter 8) that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the functions and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The *elaboration phase* encompasses the communication and modeling activities of the generic process model (Figure 4.7). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the analysis model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [Arl02] that represents a "first cut" executable system.[15] The architectural baseline demonstrates the viability of the

15 It is important to note that the architectural baseline is not a prototype in that it is not thrown away. Rather, the baseline is fleshed out during the next UP phase.

architecture but does not provide all features and functions required to use the system. In addition, the plan is carefully reviewed at the culmination of the elaboration phase to ensure that scope, risks, and delivery dates remain reasonable. Modifications to the plan are often made at this time.

WebRef

An interesting discussion of the UP in the context of agile development can be found at **www.ambysoft.com/unifiedprocess/agileUP.html**.

The *construction phase* of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, analysis and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in source code. As components are being implemented, unit tests[16] are designed and executed for each. In addition, integration activities (component assembly and integration testing) are conducted. Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

The *transition phase* of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing, and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The *production phase* of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

A software engineering workflow is distributed across all UP phases. In the context of UP, a *workflow* is analogous to a task set (described in Chapter 3). That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks. It should be noted that not every task identified for a UP workflow is conducted for every software project. The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

16 A comprehensive discussion of software testing (including *unit tests*) is presented in Chapters 22 through 26).

4.4 Personal and Team Process Models

Quote:

"A person who is successful has simply formed the habit of doing things that unsuccessful people will not do."

Dexter Yager

The best software process is one that is close to the people who will be doing the work. If a software process model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the needs of the project team that is actually doing software engineering work. In an ideal setting, you would create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization. Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization. Watts Humphrey ([Hum05] and [Hum00]) argues that it is possible to create a "personal software process" and/or a "team software process." Both require hard work, training, and coordination, but both are achievable.[17]

4.4.1 Personal Software Process

WebRef

A wide array of resources for PSP can be found at **http://www.sei.cmu.edu/tsp/tools/academic/**.

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey [Hum05] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

? What framework activities are used during PSP?

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

17 It's worth noting the proponents of agile software development (Chapter 5) also argue that the process should remain close to the team. They propose an alternative method for achieving this.

High-level design review. Formal verification methods (Appendix 3) are applied to uncover errors in the design. Metrics are maintained for important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

KEY POINT

PSP emphasizes the need to record and analyze the types of errors you make, so that you can develop strategies to eliminate them.

PSP stresses the need for you to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.

PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97]. However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach. PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high. The required level of measurement is culturally difficult for many software people.

Can PSP be used as an effective software process at a personal level? The answer is an unequivocal "yes." But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

4.4.2 Team Software Process

WebRef

Information on building high-performance teams using TSP and PSP can be obtained at **www.sei.cmu.edu/tsp/**.

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a *Team Software Process* (TSP). The goal of TSP is to build a "self-directed" project team that organizes itself to produce high-quality software. Humphrey [Hum98] defines the following objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.

- Accelerate software process improvement by making CMM[18] level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

ADVICE

To form a self-directed team, you must collaborate well internally and communicate well externally.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: *project launch, high-level design, implementation, integration and test,* and *postmortem*. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

KEY POINT

TSP scripts define elements of the team process and activities that occur within the process.

TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. "Scripts" define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

TSP recognizes that the best software teams are self-directed.[19] Team members set project objectives, adapt the process to meet their needs, control the project schedule, and through measurement and analysis of the metrics collected, work continually to improve the team's approach to software engineering.

Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality. The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

4.5 Process Technology

One or more of the process models discussed in the preceding sections must be adapted for use by a software team. To accomplish this, *process technology tools* have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.

18 The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 37.

19 In Chapter 5 we discuss the importance of "self-organizing" teams as a key element in agile software development.

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities discussed in Chapter 3. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

SOFTWARE TOOLS

Process Modeling Tools

Objective: If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* or *process management* tools) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the actions and work tasks that are required to perform it. Process modeling tools provide links to other tools that provide support to defined process activities.

Mechanics: Tools in this category allow a team to define the elements of a unique process model (actions, tasks, work products, QA points), provide detailed guidance on the content or description of each process element, and then manage the process as it is conducted. In some cases, the process technology tools incorporate standard project management tasks such as estimating, scheduling, tracking, and control.

Representative tools:[20]

Igrafx Process Tools—tools that enable a team to map, measure, and model the software process (**http://www.igrafx.com/**)

Adeptia BPM Server—designed to manage, automate, and optimize business processes (**www.adeptia.com**)

ALM Studio Suite—a collection of tools with a heavy emphasis on the management of communication and modeling activities (**http://www.kovair.com/**)

4.6 PRODUCT AND PROCESS

If the process is weak, the end product will undoubtedly suffer. But an obsessive overreliance on process is also dangerous. In a brief essay written many years ago, Margaret Davis [Dav95a] makes timeless comments on the duality of product and process:

> About every ten years give or take five, the software community redefines "the problem" by shifting its focus from product issues to process issues. Thus, we have

20 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

embraced structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process) [followed by object-oriented methods, followed by agile software development].

While the natural tendency of a pendulum is to come to rest at a point midway between two extremes, the software community's focus constantly shifts because new force is applied when the last swing fails. These swings are harmful in and of themselves because they confuse the average software practitioner by radically changing what it means to perform the job let alone perform it well. The swings also do not solve "the problem" for they are doomed to fail as long as product and process are treated as forming a dichotomy instead of a duality.

There is precedence in the scientific community to advance notions of duality when contradictions in observations cannot be fully explained by one competing theory or another. The dual nature of light, which seems to be simultaneously particle and wave, has been accepted since the 1920s when Louis de Broglie proposed it. I believe that the observations we can make on the artifacts of software and its development demonstrate a fundamental duality between product and process. You can never derive or understand the full artifact, its context, use, meaning, and worth if you view it as only a process or only a product.

All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result in a representation or instance that can be used or appreciated either by more than one person, used over and over, or used in some other context not considered. That is, we derive feelings of satisfaction from reuse of our products by ourselves or others.

Thus, while the rapid assimilation of reuse goals into software development potentially increases the satisfaction software practitioners derive from their work, it also increases the urgency for acceptance of the duality of product and process. Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity. Taking one view over the other dramatically reduces the opportunities for reuse and, hence, loses the opportunity for increasing job satisfaction.

People derive as much (or more) satisfaction from the creative process as they do from the end product. An artist enjoys the brush strokes as much as the framed result. A writer enjoys the search for the proper metaphor as much as the finished book. As creative software professional, you should also derive as much satisfaction from the process as the end product. The duality of product and process is one important element in keeping creative people engaged as software engineering continues to evolve.

4.7 Summary

Prescriptive process models have been applied for many years in an effort to bring order and structure to software development. Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment.

Sequential process models, such as the waterfall and V-models, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (e.g., continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply across all software engineering activities—from concept development to long-term system maintenance.

The concurrent process model allows a software team to represent iterative and concurrent elements of any process model. Specialized models include the component-based model that emphasizes component reuse and assembly; the formal methods model that encourages a mathematically based approach to software development and verification; and the aspect-oriented model that accommodates crosscutting concerns spanning the entire system architecture. The Unified Process is a "use case driven, architecture-centric, iterative and incremental" software process designed as a framework for UML methods and tools.

Personal and team models for the software process have been proposed. Both emphasize measurement, planning, and self-direction as key ingredients for a successful software process.

Problems and Points to Ponder

4.1. Provide three examples of software projects that would be amenable to the waterfall model. Be specific.

4.2. Provide three examples of software projects that would be amenable to the prototyping model. Be specific.

4.3. What process adaptations are required if the prototype will evolve into a delivery system or product?

4.4. Provide three examples of software projects that would be amenable to the incremental model. Be specific.

4.5. As you move outward along the spiral process flow, what can you say about the software that is being developed or maintained?

4.6. Is it possible to combine process models? If so, provide an example.

4.7. The concurrent process model defines a set of "states." Describe what these states represent in your own words, and then indicate how they come into play within the concurrent process model.

4.8. What are the advantages and disadvantages of developing software in which quality is "good enough"? That is, what happens when we emphasize development speed over product quality?

4.9. Provide three examples of software projects that would be amenable to the component-based model. Be specific.

4.10. It is possible to prove that a software component and even an entire program is correct. So why doesn't everyone do this?

4.11. Are the Unified Process and UML the same thing? Explain your answer.

Further Readings and Information Sources

Most of the software engineering books discussed in the *Further Readings* section of Chapter 2 address prescriptive process models in some detail.

Cynkovic and Larsson (*Building Reliable Component-Based Systems,* Addison-Wesley, 2002) and Heineman and Council (*Component-Based Software Engineering,* Addison-Wesley, 2001) describe the process required to implement component-based systems. Jacobson and Ng (*Aspect-Oriented Software Development with Use Cases,* Addison-Wesley, 2005) and Filman and his colleagues (*Aspect-Oriented Software Development,* Addison-Wesley, 2004) discuss the unique nature of the aspect-oriented process. Monin and Hinchey (*Understanding Formal Methods,* Springer, 2003) present a worthwhile introduction, and Boca and his colleagues (*Formal Methods,* Springer, 2009) discuss the state of the art and new directions.

Books by Kenett and Baker (*Software Process Quality: Management and Control,* Marcel Dekker, 1999) and Chrissis, Konrad, and Shrum (*CMMI for Development: Guidelines for Process Integration and Product Improvement,* 3rd ed., Addison-Wesley, 2011) consider how quality management and process design are intimately connected to one another.

In addition to Jacobson, Rumbaugh, and Booch's seminal book on the Unified Process [Jac99], books by Shuja and Krebs (*IBM Rational Unified Process Reference and Certification Guide,* IBM Press, 2008), Arlow and Neustadt (*UML 2 and the Unified Process,* Addison-Wesley, 2005), Kroll and Kruchten (*The Rational Unified Process Made Easy,* Addison-Wesley, 2003), and Farve (*UML and the Unified Process,* IRM Press, 2003) provide excellent complementary information. Gibbs (*Project Management with the IBM Rational Unified Process,* IBM Press, 2006) discusses project management within the context of the UP. Dennis, Wixom, and Tegarden (*Systems Analysis and Design with UML,* 4th ed., Wiley, 2012) tackles programming and business process modeling as it relates to UP.

A wide variety of information sources on software process models are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

5 Agile Development

Key Concepts

acceptance tests . . . 75
agile alliance. 70
agile process. 69
Agile Unified Process 82
agility 68
agility principles . . . 70
cost of change. 68
Dynamic Systems Development Method (DSDM) 79

In 2001, Kent Beck and 16 other noted software developers, writers, and consultants [Bec01] (referred to as the "Agile Alliance") signed the "Manifesto for Agile Software Development." It stated:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> *Individuals and interactions* over processes and tools
>
> *Working software* over comprehensive documentation
>
> *Customer collaboration* over contract negotiation
>
> *Responding to change* over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

Quick Look

What is it? Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

Who does it? Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

Why is it important? The modern business environment that spawns computer-based systems and software products is fast-paced and ever-changing. Agile software engineering represents a reasonable alternative to conventional software engineering for certain classes of software and certain types of software projects. It has been demonstrated to deliver successful systems quickly.

What are the steps? Agile development might best be termed "software engineering lite." The basic framework activities—communication, planning, modeling, construction, and deployment—remain. But they morph into a minimal task set that pushes the project team toward construction and delivery (some would argue that this is done at the expense of problem analysis and solution design).

What is the work product? Both the customer and the software engineer have the same view—the only really important work product is an operational "software increment" that is delivered to the customer on the appropriate commitment date.

How do I ensure that I've done it right? If the agile team agrees that the process works, and the team produces deliverable software increments that satisfy the customer, you've done it right.

Extreme Programming (XP) 72
Industrial XP 72
pair programming 75
politics of agile development 71
project velocity 73
refactoring 74
Scrum 78
spike solution 74
XP story 72

A manifesto is normally associated with an emerging political movement—one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that's exactly what agile development is all about.

Although the underlying ideas that guide agile development have been with us for many years, it has been less than two decades since these ideas have crystallized into a "movement." In essence, agile[1] methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefits, but it is not applicable to all projects, all products, all people, and all situations. It is also *not* antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work.

KEY POINT

Agile development does not mean no documents are created, it means only creating documents that will be referred to later in the development process.

In the modern economy, it is often difficult or impossible to predict how a computer-based system (e.g., a mobile application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to define requirements fully before the project begins. You must be agile enough to respond to a fluid business environment.

Fluidity implies change, and change is expensive—particularly if it is uncontrolled or poorly managed. One of the most compelling characteristics of the agile approach is its ability to reduce the costs of change through the software process.

Does this mean that a recognition of challenges posed by modern realities causes you to discard valuable software engineering principles, concepts, methods, and tools? Absolutely not! Like all engineering disciplines, software engineering continues to evolve. It can be adapted easily to meet the challenges posed by a demand for agility.

Quote:

"Agility: 1, everything else: 0."

Tom DeMarco

In a thought-provoking book on agile software development, Alistair Cockburn [Coc02] argues that the prescriptive process models introduced in Chapter 4 have a major failing: *they forget the frailties of the people who build computer software*. Software engineers are not robots. They exhibit great variation in working styles; significant differences in skill level, creativity, orderliness, consistency, and spontaneity. Some communicate well in written form, others do not. Cockburn argues that process models can "deal with people's common weaknesses with [either] discipline or tolerance" and that most prescriptive process models choose discipline. He states: "Because consistency in action is a human weakness, high discipline methodologies are fragile."

If process models are to work, they must provide a realistic mechanism for encouraging the discipline that is necessary, or they must be characterized in a manner that shows "tolerance" for the people who do software engineering work. Invariably, tolerant practices are easier for software people to adopt and sustain, but (as Cockburn admits) they may be less productive. Like most things in life, trade-offs must be considered.

1 Agile methods are sometimes referred to as *light methods* or *lean methods*.

5.1 What Is Agility?

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] provides a useful discussion:

> *Agility* has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.

ADVICE

Don't make the mistake of assuming that agility gives you license to hack out solutions. A process is required and discipline is essential.

But agility is more than an effective response to change. It also encompasses the philosophy espoused in the manifesto noted at the beginning of this chapter. It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile. It emphasizes rapid delivery of operational software and deemphasizes the importance of intermediate work products (not always a good thing); it adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects; it recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.

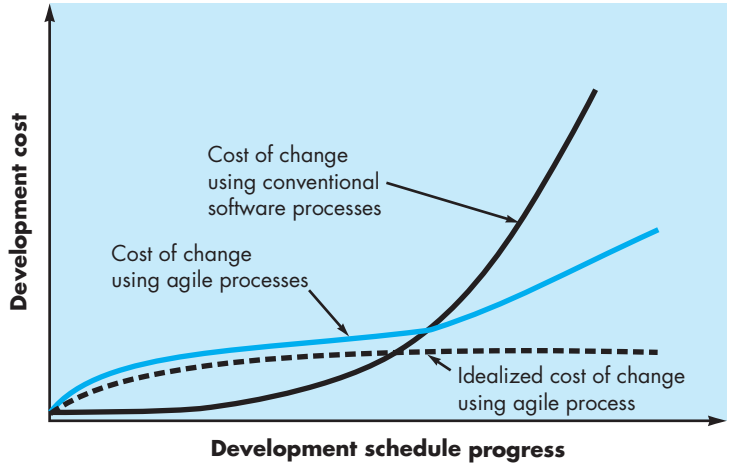
Agility can be applied to any software process. However, to accomplish this, it is essential that the process be designed in a way that allows the project team to adapt tasks and to streamline them, conduct planning in a way that understands the fluidity of an agile development approach, eliminate all but the most essential work products and keep them lean, and emphasize an incremental delivery strategy that gets working software to the customer as rapidly as feasible for the product type and operational environment.

5.2 Agility and the Cost of Change

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 5.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modified, a list of functions may be extended, or a written

FIGURE 5.1

Change costs as a function of time in development



Quote:

"Agility is dynamic, content specific, aggressively change embracing, and growth oriented."

Steven Goldman et al.

specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

KEY POINT

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 5.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed later in this chapter), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

5.3 What Is an Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

WebRef

A comprehensive collection of articles on the agile process can be found at **http://www.agilemodeling.com/.**

2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability*? The answer, as we have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable*.

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally*. To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

5.3.1 Agility Principles

KEY POINT

Although agile processes embrace change, it is still important to examine the reasons for change.

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.

ADVICE

Working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

5.3.2 The Politics of Agile Development

ADVICE

You don't have to choose between agility and software engineering. Rather, define a software engineering approach that is agile.

There has been considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making.

No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 5.4), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

If you have further interest, see [Hig01], [Hig02a], and [DeM02] for an entertaining summary of other important technical and political issues.

5.4 Extreme Programming

WebRef

An award-winnng "process simulation game" that includes an XP process module can be found at **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html.**

In order to illustrate an agile process in a bit more detail, we'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. A variant of XP, called *Industrial XP* (IXP), refines XP and targets the agile process specifically for use within large organizations [Ker05].

5.4.1 The XP Process

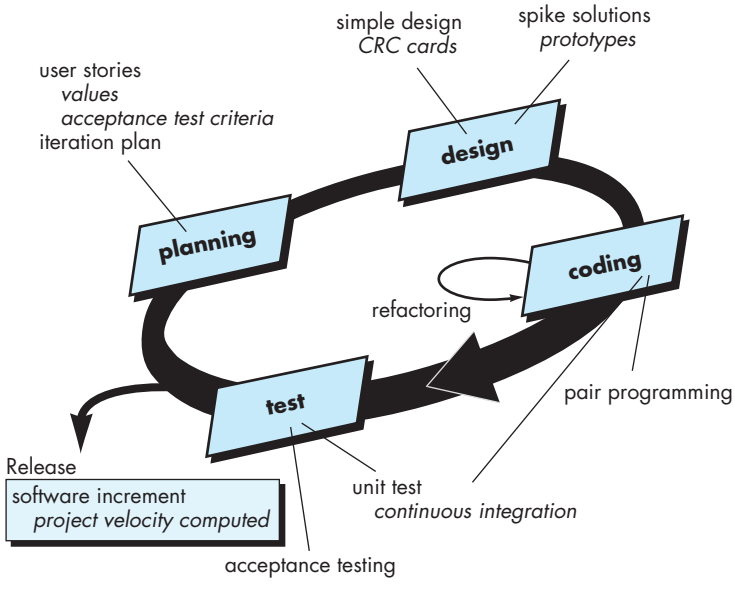
? What is an XP "story"?

Extreme Programming uses an object-oriented approach (Appendix 2) as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Figure 5.2 illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity. Key XP activities are summarized in the paragraphs that follow.

Planning. The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members

FIGURE 5.2

The Extreme Programming process



WebRef

A worthwhile XP "planning game" can be found at: **http://csis.pace.edu/~bergin/xp/planninggame.html.**

of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built. Each *story* (similar to use cases described in Chapter 8) is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.[2] Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic *commitment* (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

KEY POINT

Project velocity is a subtle measure of team productivity.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, *project velocity* is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an overcommitment has been made for all stories across the entire development project. If an overcommitment occurs, the content of releases is modified or end delivery dates are changed.

ADVICE

XP deemphasizes the importance of design. Not everyone agrees. In fact, there are times when design should be emphasized.

As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.[3]

XP encourages the use of CRC cards (Chapter 10) as an effective mechanism for thinking about the software in an object-oriented context. CRC

2 The value of a story may also be dependent on the presence of another story.

3 These design guidelines should be followed in every software engineering method, although there are times when sophisticated design notation and terminology may get in the way of simplicity.

WebRef
Refactoring techniques and tools can be found at: **www.refactoring.com.**

(class-responsibility-collaborator) cards identify and organize the object-oriented classes[4] that are relevant to the current software increment. The XP team conducts the design exercise using a process similar to the one described in Chapter 10. The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution*, the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem.

KEY POINT
Refactoring improves the internal structure of a design (or source code) without changing its external functionality or behavior.

XP encourages *refactoring*—a construction technique that is also a design technique. Fowler [Fow00] describes refactoring in the following manner:

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [and modify/simplify the internal design] that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

Because XP design uses virtually no notation and produces few, if any, work products other than CRC cards and spike solutions, design is viewed as a transient artifact that can and should be continually modified as construction proceeds. The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design" [Fow00]. It should be noted, however, that the effort required for refactoring can grow dramatically as the size of an application grows.

A central notion in XP is that design occurs both before *and after* coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

WebRef
Useful information on XP can be obtained at **www.xprogramming.com.**

Coding. After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).[5] Once the unit test[6] has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added

4 Object-oriented classes are discussed in Appendix 2, in Chapter10, and throughout Part 2 of this book.

5 This approach is analogous to knowing the exam questions before you begin to study. It makes studying much easier by focusing attention only on the questions that will be asked.

6 Unit testing, discussed in detail in Chapter 22, focuses on an individual software component, exercising the component's interface, data structures, and functionality in an effort to uncover errors that are local to the component.

(KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

What is pair programming?

A key concept during the coding activity (and one of the most talked-about aspects of XP) is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.[7]

ADVICE

Many software teams are populated by individualists. You'll have to work to change that culture if pair programming is to work effectively.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment (Chapter 22) that helps to uncover errors early.

How are unit tests used in XP?

Testing. The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly). This encourages a regression testing strategy (Chapter 22) whenever code is modified (which is often, given the XP refactoring philosophy).

As the individual unit tests are organized into a "universal testing suite" [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

KEY POINT

XP acceptance tests are derived from user stories.

XP *acceptance tests*, also called *customer tests,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

5.4.2 Industrial XP

What new practices are appended to XP to create IXP?

Joshua Kerievsky [Ker05] describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that

7 Pair programming has become so widespread throughout the software community that *The Wall Street Journal* [Wal12] ran a front-page story about the subject.

are designed to help ensure that an XP project works successfully for significant projects within a large organization:

Readiness assessment. The IXP team ascertains whether all members of the project community (e.g., stakeholders, developers, management) are on board, have the proper environment established, and understand the skill levels involved.

Project community. The IXP team determines whether the right people, with the right skills and training have been staged for the project. The "community" encompasses technologists and other stakeholders.

Project chartering. The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization.

Test-driven management. An IXP team establishes a series of measurable "destinations" [Ker05] that assess progress to date and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives. An IXP team conducts a specialized technical review (Chapter 20) after a software increment is delivered. Called a *retrospective*, the review examines "issues, events, and lessons-learned" [Ker05] across a software increment and/or the entire software release.

Continuous learning. The IXP team is encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher-quality product.

Quote:

"Ability is what you're capable of doing. Motivation determines what you do. Attitude determines how well you do it."

Lou Holtz

In addition to the six new practices discussed, IXP modifies a number of existing XP practices and redefines certain roles and responsibilities to make them more amenable to significant projects for large organizations. For further discussion of IXP, visit **http://industrialxp.org.**

SafeHome

Considering Agile Software Development

The scene: Doug Miller's office.

The Players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation:
(A knock on the door, Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure Jamie, what's up?

Jamie: We've been thinking about our process discussion yesterday . . . you know, what process we're going to choose for this new *SafeHome* project.

Doug: And?

Vinod: I was talking to a friend at another company, and he was telling me about Extreme Programming. It's an agile process model . . . heard of it?

Doug: Yeah, some good, some bad.

Jamie: Well, it sounds pretty good to us. Lets you develop software really fast, uses something called pair programming to do real-time quality checks . . . it's pretty cool, I think.

Doug: It does have a lot of really good ideas. I like the pair-programming concept, for instance, and the idea that stakeholders should be part of the team.

Jamie: Huh? You mean that marketing will work on the project team with us?

Doug (nodding): They're a stakeholder, aren't they?

Jamie: Jeez . . . they'll be requesting changes every five minutes.

Vinod: Not necessarily. My friend said that there are ways to "embrace" changes during an XP project.

Doug: So you guys think we should use XP?

Jamie: It's definitely worth considering.

Doug: I agree. And even if we choose an incremental model as our approach, there's no reason why we can't incorporate much of what XP has to offer.

Vinod: Doug, before you said "some good, some bad." What was the bad?

Doug: The thing I don't like is the way XP downplays analysis and design . . . sort of says that writing code is where the action is . . .

(The team members look at one another and smile.)

Doug: So you agree with the XP approach?

Jamie (speaking for both): Writing code is what we do, Boss!

Doug (laughing): True, but I'd like to see you spend a little less time coding and then recoding and a little more time analyzing what has to be done and designing a solution that works.

Vinod: Maybe we can have it both ways, agility with a little discipline.

Doug: I think we can, Vinod. In fact, I'm sure of it.

5.5 Other Agile Process Models

Quote:

"Our profession goes through methodologies like a 14-year-old goes through clothing."

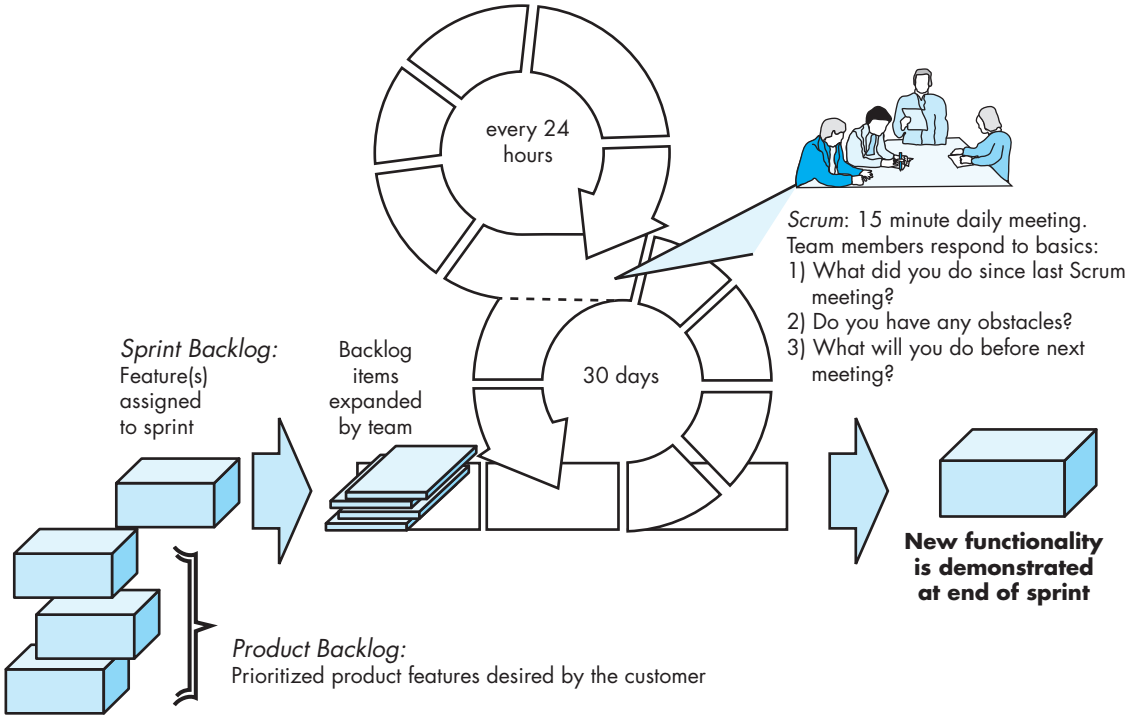
Stephen Hawrysh and Jim Ruprecht

The history of software engineering is littered with dozens of obsolete process descriptions and methodologies, modeling methods and notations, tools, and technology. Each flared in notoriety and was then eclipsed by something new and (purportedly) better. With the introduction of a wide array of agile process models—each contending for acceptance within the software development community—the agile movement is following the same historical path.[8]

As we noted in the last section, the most widely used of all agile process models is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. In this section, we present a brief overview of four common agile methods: Scrum, DSSD, Agile Modeling (AM), and Agile Unified Process (AUP).

8 This is not a bad thing. Before one or more models or methods are accepted as a de facto standard, all must contend for the hearts and minds of software engineers. The "winners" evolve into best practice, while the "losers" either disappear or merge with the winning models.

FIGURE 5.3 Scrum process flow



5.5.1 Scrum

WebRef
Useful Scrum information and resources can be found at **www.controlchaos.com.**

Scrum (the name is derived from an activity that occurs during a rugby match)[9] is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle [Sch01b].

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern (discussed in the following paragraph) called a *sprint*. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in Figure 5.3.

9 A group of players forms around the ball and the teammates work together (sometimes violently!) to move the ball downfield.

Scrum emphasizes the use of a set of software process patterns [Noy02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development activities:

KEY POINT

Scrum incorporates a set of process patterns that emphasize project priorities, compartmentalized work units, communication, and frequent customer feedback.

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box[10] (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings—are short (typically 15-minute) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members [Noy02]:

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a *Scrum master*, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Beedle and his colleagues [Bee99] present a comprehensive discussion of these patterns in which they state: "Scrum assumes up-front the existence of chaos . . ." The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

5.5.2 Dynamic Systems Development Method

WebRef

Useful resources for DSDM can be found at **www.dsdm.org.**

The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental

10 A *time-box* is a project management term (see Part 4 of this book) that indicates a period of time that has been allocated to accomplish some task.

prototyping in a controlled project environment" [CCS02]. The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The DSDM Consortium (**www.dsdm.org**) is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the *DSDM life cycle,* that begins with a *feasibility study* that establishes basic business requirements and constraints and is followed by a *business study* that identifies functional and information requirements. DSDM then defines three different iterative cycles:

KEY POINT

DSDM is a process framework that can adopt the tactics of another agile approach such as XP.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. (Note: All DSDM prototypes are intended to evolve into the deliverable application.) The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—revisits prototypes built during the functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, the functional model iteration and the design and build iteration occur concurrently.

Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

DSDM can be combined with XP (Section 5.4) to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments.

5.5.3 Agile Modeling

WebRef

Comprehensive information on agile modeling can be found at: **www.agilemodeling.com.**

There are many situations in which software engineers must build large, business-critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built. But in some cases, it can be daunting to manage the volume of notation

required, the degree of formalism suggested, the sheer size of the models for large projects, and the difficulty in maintaining the model(s) as changes occur. Is there an agile approach to software engineering modeling that might provide some relief?

At "The Official Agile Modeling Site," Scott Ambler [Amb02a] describes *agile modeling* (AM) in the following manner:

> Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

Quote:

"I was in the drugstore the other day trying to get a cold medication . . . Not easy. There's an entire wall of products you need. You stand there going, Well, this one is quick acting but this is long lasting ... Which is more important, the present or the future?"

Jerry Seinfeld

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Although AM suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are [Amb02a]:

Model with a purpose. A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

Use multiple models. There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

ADVICE

"Traveling light" is an appropriate philosophy for all software engineering work. Build only those models that provide value . . . no more, no less.

Travel light. As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler [Amb02a] notes that "Every time you decide to keep a model you trade off agility for the convenience of having that information available to your team in an abstract manner (hence potentially enhancing communication within your team as well as with project stakeholders)."

Content is more important than representation. Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.

Adapt locally. The modeling approach should be adapted to the needs of the agile team.

A major segment of the software engineering community has adopted the Unified Modeling Language (UML)[11] as the preferred method for representing analysis and design models. The Unified Process (Chapter 4) has been developed to provide a framework for the application of UML. Scott Ambler [Amb06] has developed a simplified version of the UP that integrates his agile modeling philosophy.

5.5.4 Agile Unified Process

The *Agile Unified Process* (AUP) adopts a "serial in the large" and "iterative in the small" [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities [Amb06]:

- *Modeling*. UML representations of the business and problem domains are created. However, to stay agile, these models should be "just barely good enough" [Amb06] to allow the team to proceed.
- *Implementation*. Models are translated into source code.
- *Testing*. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- *Deployment*. Like the generic process activity discussed in Chapters 3, deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- *Configuration and project management*. In the context of AUP, configuration management (Chapter 29) addresses change management, risk management, and the control of any persistent work products[12] that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

11 A brief tutorial on UML is presented in Appendix 1.

12 A *persistent work product* is a model or document or test case produced by the team that will be kept for an indeterminate period of time. It will *not* be discarded once the software increment is delivered.

- *Environment management.* Environmental management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Although the AUP has historical and technical connections to the Unified Modeling Language, it is important to note that UML modeling can be used in conjunction with any of the agile process models described in this chapter.

SOFTWARE TOOLS

Agile Development

Objective: The objective of agile development tools is to assist in one or more aspects of agile development with an emphasis on facilitating the rapid generation of operational software. These tools can also be used when prescriptive process models (Chapter 4) are applied.

Mechanics: Tool mechanics vary. In general, agile tool sets encompass automated support for project planning, use case development and requirements gathering, rapid design, code generation, and testing.

Representative tools:[13]

Note: Because agile development is a hot topic, most software tools vendors purport to sell tools that support the agile approach. The tools noted here have characteristics that make them particularly useful for agile projects.

OnTime, developed by Axosoft (**www.axosoft.com**), provides agile process management support for various technical activities within the process.

Ideogramic UML, developed by Ideogramic (**http://ideogramic-uml.software.informer.com/**) is a UML tool set specifically developed for use within an agile process.

Together Tool Set, distributed by Borland (**www.borland.com**), provides a tools suite that supports many technical activities within XP and other agile processes.

5.6 A TOOL SET FOR THE AGILE PROCESS

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops."

KEY POINT

The "tool set" that supports agile processes focuses more on people issues than it does on technology issues.

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04] or modern social networking techniques) that provides information and coordination among agile developers. Active communication is achieved via the team dynamics (e.g., pair programming), while

13 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

passive communication is achieved by "information radiators" (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or "graphs of tests created versus passed . . . other agile tools are using to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)" [Coc04].

Are any of these things really tools? They are, if they facilitate the work performed by an agile team member and enhance the quality of the end product.

5.7 Summary

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile—to define maneuverable, adaptive, lean processes that can accommodate the needs of modern business.

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform, communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity, and an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Extreme programming (XP) is the most widely used agile process. Organized as four framework activities—planning, design, coding, and testing—XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and then prioritized by stakeholders.

Other agile process models also stress human collaboration and team self-organization, but define their own framework activities and select different points of emphasis. For example, Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project. The Dynamic Systems Development Method (DSDM) advocates the use of time-box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment. Agile modeling (AM) suggests that modeling is essential for all systems, but that the complexity, type, and size of the model must be tuned to the software to be built. The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building software.

Problems and Points to Ponder

5.1. Reread the "Manifesto for Agile Software Development" at the beginning of this chapter. Can you think of a situation in which one or more of the four "values" could get a software team into trouble?

5.2. Describe agility (for software projects) in your own words.

5.3. Why does an iterative process make it easier to manage change? Is every agile process discussed in this chapter iterative? Is it possible to complete a project in just one iteration and still be agile? Explain your answers.

5.4. Could each of the agile processes be described using the generic framework activities noted in Chapter 3? Build a table that maps the generic activities into the activities defined for each agile process.

5.5. Try to come up with one more "agility principle" that would help a software engineering team become even more maneuverable.

5.6. Select one agility principle noted in Section 5.3.1 and try to determine whether each of the process models presented in this chapter exhibits the principle. [Note: We have presented an overview of these process models only, so it may not be possible to determine whether a principle has been addressed by one or more of the models, unless you do additional research (which is not required for this problem).]

5.7. Why do requirements change so much? After all, don't people know what they want?

5.8. Most agile process models recommend face-to-face communication. Yet today, members of a software team and their customers may be geographically separated from one another. Do you think this implies that geographical separation is something to avoid? Can you think of ways to overcome this problem?

5.9. Write an XP user story that describes the "favorite places" or "favorites" feature available on most Web browsers.

5.10. What is a spike solution in XP?

5.11. Describe the XP concepts of refactoring and pair programming in your own words.

5.12. Using the process pattern template presented in Chapter 3, develop a process pattern for any one of the Scrum patterns presented in Section 5.5.1.

5.13. Visit the Official Agile Modeling site and make a complete list of all core and supplementary AM principles.

5.14. The tool set proposed in Section 5.6 supports many of the "soft" aspects of agile methods. Since communication is so important, recommend an actual tool set that might be used to enhance communication among stakeholders on an agile team.

Further Readings and Information Sources

The overall philosophy and underlying principles of agile software development are considered in-depth in many of the books referenced in the body of this chapter. In addition, books by Pichler (*Agile Project Management with Scrum: Creating Products that Customers Love,* Addison-Wesley, 2010), Highsmith (*Agile Project Management: Creating Innovative Products,* 2nd ed. Addison-Wesley, 2009), Shore and Chromatic (*The Art of Agile Development,* O'Reilly Media, 2008), Hunt (*Agile Software Construction,* Springer, 2005), and Carmichael and Haywood (*Better Software Faster,* Prentice Hall, 2002) present useful discussions of the subject. Aguanno (*Managing Agile Projects,* Multi-Media Publications, 2005), and Larman (*Agile and*

Iterative Development: A Manager's Guide, Addison-Wesley, 2003) present a management overview and consider project management issues. Highsmith (*Agile Software Development Ecosystems,* Addison-Wesley, 2002) presents a survey of agile principles, processes, and practices. A worthwhile discussion of the delicate balance between agility and discipline is presented by Booch and his colleagues (*Balancing Agility and Discipline,* Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship,* Prentice Hall, 2009) presents the principles, patterns, and practices required to develop "clean code" in an agile software engineering environment. Leffingwell (*Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, Addison-Wesley, 2011) and (*Scaling Software Agility: Best Practices for Large Enterprises,* Addison-Wesley, 2007) discusses strategies for scaling up agile practices for large projects. Lippert and Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully,* Wiley, 2006) discuss the use of refactoring when applied in large, complex systems. Stamelos and Sfetsos (*Agile Software Development Quality Assurance,* IGI Global, 2007) discuss SQA techniques that conform to the agile philosophy.

Dozens of books have been written about Extreme Programming over the past decade. Beck (*Extreme Programming Explained: Embrace Change,* 2nd ed., Addison-Wesley, 2004) remains the definitive treatment of the subject. In addition, Jeffries and his colleagues (*Extreme Programming Installed,* Addison-Wesley, 2000), Succi and Marchesi (*Extreme Programming Examined,* Addison-Wesley, 2001), Newkirk and Martin (*Extreme Programming in Practice,* Addison-Wesley, 2001), and Auer and his colleagues (*Extreme Programming Applied: Play to Win,* Addison-Wesley, 2001) provide a nuts-and-bolts discussion of XP along with guidance on how best to apply it. McBreen (*Questioning Extreme Programming,* Addison-Wesley, 2003) takes a critical look at XP, defining when and where it is appropriate. An in-depth consideration of pair programming is presented by McBreen (*Pair Programming Illuminated,* Addison-Wesley, 2003).

Kohut (*Professional Agile Development Process: Real World Development Using SCRUM,* Wrox, 2013), Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012), Larman and Vodde (*Scaling Lean and Agile Development: Thinking and Organizational Tools for Large Scale Scrum,* Addison-Wesley, 2008), and Schwaber (*The Enterprise and Scrum,* Microsoft Press, 2007) discuss the use of Scrum for projects that have a major business impact. The nuts and bolts of Scrum are discussed by Cohn (*Succeeding with Agile*, Addison-Wesley, 2009), and Schwaber and Beedle (*Agile Software Development with SCRUM,* Prentice-Hall, 2001). Worthwhile treatments of DSDM have been written by the DSDM Consortium (*DSDM: Business Focused Development,* 2nd ed., Pearson Education, 2003) and Stapleton (*DSDM: The Method in Practice,* Addison-Wesley, 1997).

Books by Ambler and Lines (*Disciplined Agile Delivery: A Practitioner's Guide to Agile Delivery in the Enterprise*, IBM Press, 2012) and Poppendieck and Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers,* Addison-Wesley, 2003) provide guidelines for managing and controlling agile projects. Ambler and Jeffries (*Agile Modeling,* Wiley, 2002) discuss AM in some depth.

A wide variety of information sources on agile software development are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the agile process can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

6

Human Aspects of Software Engineering

Key Concepts

agile teams..... 93
cloud computing ... 97
collaborative development environments (CDEs) 98
global teams..... 99
jelled team 90
psychology 89
roles 89
social media 95
team attributes.... 90
team structures.... 92
team toxicity 91
traits 88
XP team..... 94

In a special issue of *IEEE Software*, the guest editors [deS09] make the following observation:

> Software engineering has an abundance of techniques, tools, and methods designed to improve both the software development process and the final product. Technical improvements continue to emerge and yield encouraging results. However, software isn't simply a product of the appropriate technical solutions applied inappropriate technical ways. Software is developed by people, used by people, and supports interaction among people. As such, human characteristics, behavior, and cooperation are central to practical software development.

Throughout the chapters that follow this one, we'll discuss the "techniques, tools, and methods" that will result in the creation of a successful software product. But before we do, it is essential to understand that without skilled and motivated people, success is unlikely.

Quick Look

What is it? We all tend to get caught up in the latest programming language, the best new design methods, the most fashionable agile process, or a just released whiz-bang software tool. But at the end of the day, *people* build computer software. And for that reason, the human aspects of software engineering often have as much to do with the success of a project as the latest and greatest technology.

Who does it? Individuals and teams do software engineering work. In some cases, one person has much of the responsibility, but in most industry-grade software efforts, a team of people does the work.

Why is it important? A software team will be successful only if the dynamics of the team are right. Software engineers sometimes have a reputation of not playing well with others. In reality, it is essential for software engineers on a team to play well with their colleagues and with other stakeholders in the product to be built.

What are the steps? First, you have to understand the personal characteristics of a successful software engineer and then try to emulate them. Next, you should appreciate the complex psychology of software engineering work, so that you can navigate your way through a project without peril. Then, you have to understand the structure and dynamics of a software team, because team-based software engineering is common in an industry setting. Finally, you should appreciate the impact of social media, the cloud, and other collaborative tools.

What is the work product? Better insight into the people, the process, and the end product.

How do I ensure that I've done it right? Spend the time to observe how successful software engineers do their work and tune your approach to take advantage of the strengths they project.

6.1 Characteristics of a Software Engineer

Quote:

Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.

Linus Torvalds

? What are the personal characteristics of an effective software engineer?

So you want to be a software engineer? Obviously, you have to master the technical stuff, learn and apply the skills required to understand the problem, design an effective solution, build the software, and test it in an effort to develop the highest quality possible. You have to manage change, communicate with stakeholders, and use appropriate tools in the appropriate situations. All of these things are discussed at length later in this book.

But there are other things that are equally important—the human aspects that will make you an effective software engineer. Erdogmus [Erd09] identifies seven traits that are present when an individual software engineer exhibits "superprofessional" behavior.

An effective software engineer has a sense of *individual responsibility*. This implies a drive to deliver on her promises to peers, stakeholders, and her management. It implies that she will do what needs to be done, when it needs to be done in an overriding effort to achieve a successful outcome.

An effective software engineer has an *acute awareness* of the needs of other members of his team, of the stakeholders that have requested a software solution to an existing problem, and the managers who have overall control over the project that will achieve that solution. He is able to observe the environment in which people work and adapt his behavior to both the environment and the people themselves.

An effective software engineer is *brutally honest*. If she sees a flawed design, she points out the flaws in a constructive but honest manner. If asked to distort facts about schedule, features, performance, or other product or project characteristics she opts to be realistic and truthful.

An effective software engineer exhibits *resilience under pressure*. As we noted previously in this book, software engineering is always on the edge of chaos. Pressure (and the chaos that can result) comes in many forms—changes in requirements and priorities, demanding stakeholders or peers, an unrealistic or overbearing manager. But an effective software engineer is able to manage the pressure so that his performance does not suffer.

An effective software engineer has a *heightened sense of fairness*. She gladly shares credit with her colleagues. She tries to avoid conflicts of interest and never acts to sabotage the work of others.

An effective software engineer exhibits *attention to detail*. This does not imply an obsession with perfection, but it does suggest that he carefully considers the technical decisions he makes on a daily basis against broader criteria (e.g., performance, cost, quality) that have been established for the product and the project.

Finally, an effective software engineer is pragmatic. She recognizes that software engineering is not a religion in which dogmatic rules must be followed, but rather a discipline that can be adapted based on the circumstances at hand.

6.2 The Psychology of Software Engineering

In a seminal paper on the psychology of software engineering, Bill Curtis and Diane Walz [Cur90] suggest a layered behavioral model for software development (Figure 6.1). At an individual level, software engineering psychology focuses on recognition of the problem to be solved, the problem-solving skills required to solve it, and the motivation to complete the solution within the constraints established by outer layers in the model. At the team and project levels, group dynamics becomes the dominating factor. Here, team structure and social factors govern success. Group communication, collaboration, and coordination are as important as the skills of an individual team member. At the outer layers, organizational behavior governs the actions of the company and its response to the business milieu.

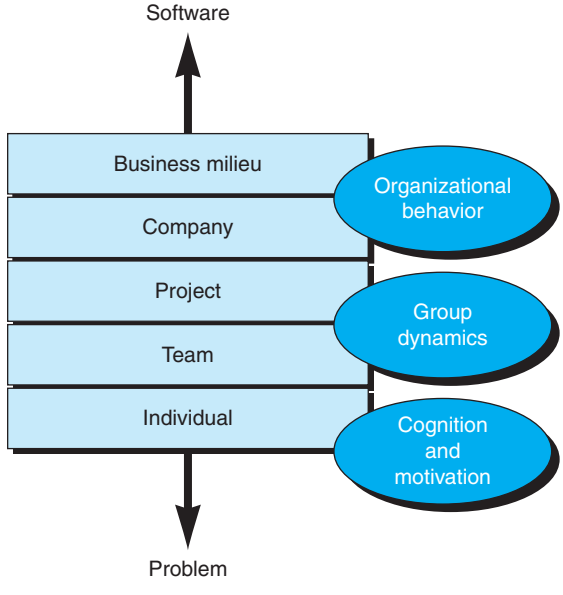
At the team level, Sawyer and his colleagues [Saw08] suggest that teams often establish artificial boundaries that reduce communication and, as a consequence, reduce the team effectiveness. They suggest a set of "boundaries spanning roles" that allow members of a software team to effectively move across team boundaries. The following roles may be assigned explicitly or can evolve naturally.

? What roles do members of a software team play?

- *Ambassador*—represents the team to outside constituencies with the intent of negotiating time and resources and gaining feedback from stakeholders.
- *Scout*—crosses the team's boundary to collect organizational information. "Scouting can include scanning about external markets, searching for new

Figure 6.1

A layers behavioral model for software engineering (adapted from [Cur90])



technologies, identifying relevant activities outside of the team and uncovering pockets of potential competition." [Saw08]

- *Guard*—protects access to the team's work products and other information artifacts.
- *Sentry*—controls the flow of information that stakeholders and others send to the team.
- *Coordinator*—focuses on communicating horizontally across the team and within the organization (e.g., discussing a specific design problem with a group of specialists within the organization).

6.3 The Software Team

In their classic book *Peopleware,* Tom DeMarco and Tim Lister [DeM98] discuss the cohesiveness of a software team:

> We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

? What is a "jelled" team?

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

There is no foolproof method for creating a jelled team. But there are attributes that are normally found in effective software teams.[1] Miguel Carrasco [Car08] suggests that an effective software team must establish a *sense of purpose*. For example, if all team members agree that the goal of the team is to develop software that will transform a product category, and as a consequence, vault their company into an industry leader, they have a strong sense of purpose. An effective team must also inculcate a *sense of involvement* that allows every member to feel that his skill set and contributions are valued.

1 Bruce Tuckman observes that successful teams go through four phases (Forming, Storming, Norming, and Performing) on their way to becoming productive (**http://www.realsoftwaredevelopment.com/7-key-attributes-of-high-performance-software-development-teams/**)

KEY POINT

An effective software team is diverse, populated by people who have a sense of purpose, involvement, trust, and improvement.

? Why is it that teams fail to jell?

An effective team should foster a *sense of trust*. Software engineers on the team should trust the skills and competence of their peers and their managers. The team should encourage a *sense of improvement*, by periodically reflecting on its approach to software engineering and looking for ways to improve their work.

The most effective software teams are diverse in the sense that they combine a variety of different skill sets. Highly skilled technologists are complemented by members who may have less technical background but are more empathetic to the needs of stakeholders.

But not all teams are effective and not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

To avoid a frenzied work environment, the team should have access to all information required to do the job. Major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built, the people doing the work, and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[2] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

Quote:

"Not every group is a team, and not every team is effective."

Glenn Parker

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverts. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last-minute deadline. Recognition of human differences, along with other guidelines presented in this section, provide a higher likelihood of creating teams that jell.

2 Technical reviews are discussed in detail in Chapter 20.

6.4 Team Structures

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes a number of project factors that should be considered when planning the structure of software engineering teams: (1) difficulty of the problem to be solved, (2) "size" of the resultant program(s) in lines of code or function points,[3] (3) time that the team will stay together (team lifetime), (4) degree to which the problem can be modularized, (5) required quality and reliability of the system to be built, (6) rigidity of the delivery date, and (7) degree of sociability (communication) required for the project.

Constantine [Con93] suggests four "organizational paradigms" for software engineering teams:

? What factors should be considered when the structure of a software team is chosen?

? What options do we have when defining the structure of a software team?

1. A *closed paradigm* structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. A *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. An *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. A *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

Quote:

"If you want to be incrementally better: Be competitive. If you want to be exponentially better: Be cooperative."

Author unknown

As a historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the *chief programmer team*. This structure was first proposed by Harlan Mills and described by Baker [Bak72]. The nucleus of the team was composed of a *senior engineer* (the chief programmer), who plans, coordinates, and reviews all technical activities of the team; *technical staff* (normally two to five people), who conduct analysis and development

3 Lines of code (LOC) and function points are measures of the size of a computer program and are discussed in Chapter 33.

activities; and a *backup engineer,* who supports the senior engineer in her activities and can replace the senior engineer with minimum loss in project continuity. The chief programmer may be served by one or more specialists (e.g., telecommunications expert, database designer), support staff (e.g., technical writers, clerical personnel), and a software librarian.

As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed *innovative anarchy.* Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization.

SafeHome

Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold onto that for a moment. I'd like to talk about how we're going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and that fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

6.5 Agile Teams

Over the past decade, agile software development (Chapter 5) has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity.

6.5.1 The Generic Agile Team

The small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in the

KEY POINT

An agile team is a self-organizing team that has autonomy to plan and make technical decisions.

preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write:

> If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job.

Quote:

"Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the [agile] team, not individuals who make up the team."

Jim Highsmith

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing*. A self-organizing team does not necessarily maintain a single team structure, but instead, uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 6.2.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

6.5.2 The XP Team

ADVICE

Keep it simple whenever you can, but recognize that continual "refactoring" can absorb significant time and resources.

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of extreme programming (XP)—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

In order to achieve effective *communication* between the agile team and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors[4] for communicating

4 In the XP context, a *metaphor* is "a story that everyone—customers, programmers, and managers—can tell about how the system works" [Bec04a].

important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

To achieve *simplicity,* the agile team designs only for immediate needs, rather than considering future needs. The intent is to create a simple design that can be easily implemented in code. If the design must be improved, it can be *refactored*[5] at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy (Chapters 22 through 26), the software (via test results) provides the agile team with feedback. The team makes use of the *unit test* as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the *user stories* or *use cases* (Chapter 9) that are implemented by the increment are used to perform acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Quote:

"XP is the answer to the question, 'How little can we do and still build great software?' "

Anonymous

Beck [Bec04a] argues that strict adherence to certain XP practices demands *courage*. A better word might be *discipline*. For example, there is often significant pressure to design for future requirements. Most software teams succumb, arguing that "designing for tomorrow" will save time and effort in the long run. An XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the XP team inculcates *respect* among its members, between other stakeholders and team members, and indirectly, for the software itself. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

6.6 The Impact of Social Media

Email, texting, and videoconferencing have become ubiquitous activities in software engineering work. But these communication mechanisms are really nothing more than modern substitutes or supplements for the face-to-face contact. Social media is different.

5 Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior. In essence, refactoring can be used to improve the efficiency, readability, or performance of a design or the code that implements a design.

Begel [Beg10] and his colleagues address the growth and application of social media in software engineering when they write:

> The social processes around software development are . . . highly dependent on engineers' abilities to find and connect with individuals who share similar goals and complementary skills, to harmonize each team member's communication and teaming preferences, to collaborate and coordinate during the entire software lifecycle, and advocate for their product's success in the marketplace.

In some ways, this "connection" can be as important as face-to-face communication. The value of social media grows as team size increases, and is magnified further when the team is geographically dispersed.

First, a social network is defined for a software project. Using the network, the software team can draw from the collective experience of team members, stakeholders, technologists, specialists, and other businesspeople who have been invited to participate in the network (if the network is private) or to any interested party (if the network is public). And it can do this whenever an issue, a question, or a problem arises. There are a number of different forms of social media and each has a place in software engineering work.

A *blog* can be used to post a series of short articles describing important aspects of a system or voicing opinions about system features or functions that are yet to be developed. It is also important to note that "software companies frequently use blogs to share technical information and opinions with their employees, and very profitably, with their customers, both internal and external." [Beg10]

Quote:

"If content is king, then conversation is queen."

John Munsell

Microblogs (e.g., Twitter) allow a member of a software engineering network to post short messages to followers who subscribe to them. Because the messages are instantaneous and can be read from all mobile platforms, dispersion of information is close to real time. This enables a software team to call an impromptu meeting if an issue arises, to ask for specialized help if a problem occurs, or to inform stakeholders about some aspect of the project.

Targeted on-line forums allow participants to post questions, opinions, case studies or any other relevant information. A technical question can be posted and within a few minutes, multiple "answers" are often available.

Social networking sites (e.g., Facebook, LinkedIn) allow degrees-of-separation connections among software developers and related technologists. This allows "friends" on a social networking site to learn about friends of friends who may have knowledge or expertise related to the application domain or problem to be solved. Specialized private networks built on the social networking paradigm can be used within an organization.

Most social media enables the formation of "communities" of users with similar interests. For example, a community of software engineers who specialize in real-time embedded systems might provide a useful way for an individual or

team working in that area to make connections that would enhance their work. As a community grows, participants discuss technology trends, application scenarios, new tools, and other software engineering knowledge. Finally, *social bookmarking sites* (e.g., Delicious, Stumble, CiteULike) allow a software engineer or team to recommend Web-based resources that may be of interest to a social media community of like-minded individuals.

It is very important to note that privacy and security issues should not be overlooked when using social media for software engineering work. Much of the work performed by software engineers may be proprietary to their employer and disclosure could be very harmful. For that reason, the distinct benefits of social media must be weighed against the treat of uncontrolled disclosure of private information.

6.7 Software Engineering Using the Cloud

Quote:

"They don't call it the Internet anymore, they call it cloud computing. I'm no longer resisting the name. Call it what you want."

Larry Ellison

Cloud computing provides a mechanism for access to all software engineering work products, artifacts, and project-related information. It runs everywhere and removes the device dependency that was once a constraint for many software projects. It allows members of a software team to conduct platform-independent, low-risk trials of new software tools and to provide feedback on those tools. It provides new avenues for distribution and testing of beta software. It provides the potential for improved approaches to content and configuration management (Chapter 29).

Because cloud computing can accomplish these things, it has the potential to influence the manner in which software engineers organize their teams, the way they do their work, the manner in which they communicate and connect, and the way software projects are managed. Software engineering information developed by one team member can be instantly available to all team members, regardless of the platform others are using or their location.

ADVICE

The cloud is a powerful repository for software engineering information, but you must be sure to consider the change control issues discussed in Chapter 29.

In essence, information dispersion speeds up and broadens dramatically. That changes the software engineering dynamic and can have a profound impact on the human aspects of software engineering.

But cloud computing in a software engineering milieu is not without risk [The13]. The cloud is dispersed over many servers and the architecture and services are often outside the control of a software team. As a consequence, there are multiple points of failure, presenting reliability and security risks. As the number of services provided by the cloud grows, the relative complexity of the software development environment also grows. Does each of these services play well with other services, possibly provided by other vendors? This presents an interoperability risk for cloud services. Finally, if the cloud becomes the development environment, services must stress usability and performance. These attributes sometime conflict with security, privacy, and reliability.

But from the human perspective, the cloud offers far more benefits than risks for software engineers. Dana Gardner [Gar09] summarizes the benefits (with a warning):

> Anything having to do with the social or collaboration aspects of software development lent themselves well to the cloud. Project management, scheduling, task lists, requirements, and defect management all suit themselves well as these are at core group functions where communications is essential to keeping projects in sync and all members of the team – wherever they are located — on literally the same page. Of course, there is a huge caveat here – if your company designs embedded software that goes into products, it is not a good candidate for the cloud: imagine getting a hold of Apple's project plans for the next version of the iPhone.

As Gardner states, one of the key benefits of the cloud is its ability to enhance the "social and collaborative aspects of software development." In the next section, you'll learn a bit more about collaborative tools.

6.8 Collaboration Tools

Fillipo Lanubile and his colleagues [Lan10] suggest that the software development environments (SDEs) of the last century have morphed into *collaborative development environments* (CDEs).[6] They state:

> Tools are essential to collaboration among team members, enabling the facilitation, automation, and control of the entire development process. Adequate tool support is especially needed in global software engineering because distance aggravates coordination and control problems, directly or indirectly, through its negative effects on communication.

Many of the tools used in a CDE are no different from the tools that are used to assist in the software engineering activities discussed in Parts 2, 3, and 4 of this book. But a worthwhile CDE also provides a set of services that are specifically designed to enhance collaborative work [Fok10]. These services include:

? What generic services are found in collaborative development environments?

- A *namespace* that allows a project team to store all work products and other information in a manner that enhances security and privacy, allowing access only to authorized individuals.
- A *calendar* for coordinating meeting and other project events.
- *Templates* that enable team members to create work products that have a consistent look and structure.
- *Metrics support* that tracks each team member's contributions in a quantitative manner.

6 The term *collaborative development environment* (CDE) was coined by Grady Booch [Boo02].

- *Communication analysis* that tracks communication across the team and isolates patterns that may imply problems or issues that need to be resolved.
- *Artifact-clustering* that organizes work products and other project artifacts in a manner that answers questions such as: "What triggered a particular change, who has discussed a specific artifact that should potentially be consulted about changes to it, and how might a [team] member's own work affect other people's work?" [Fok10].

SOFTWARE TOOLS

Collaborative Development Environments

Objective: As software development becomes global, software teams need more than development tools. They need a set of services that enable members of the team to collaborate locally and over long distances.

Mechanics: Tools and services in this category allow a team to establish mechanisms for collaborative work. A CDE will implement many or all of the services described in Section 6.6, while at the same time provide access to process management (Chapter 4) conventional software engineering tools discussed throughout this book.

Representative tools:[7]

GForge—a collaborative environment that contains both project and code management facilities **(http://gforge.com/gf/)**

OneDesk—provides a collaborative environment that creates and manages a project workspace for developers and stakeholders **(www.onedesk.com)**

Rational Team Concert—an in-depth, collaborative life-cycle management system **(http://www-01.ibm.com/software/rational/products/rtc/)**

6.9 GLOBAL TEAMS

Quote:

"More and more, in any company, managers are dealing with different cultures. Companies are going global, but the teams are being divided and scattered all over the planet."

Carlos Ghosn, Nissan

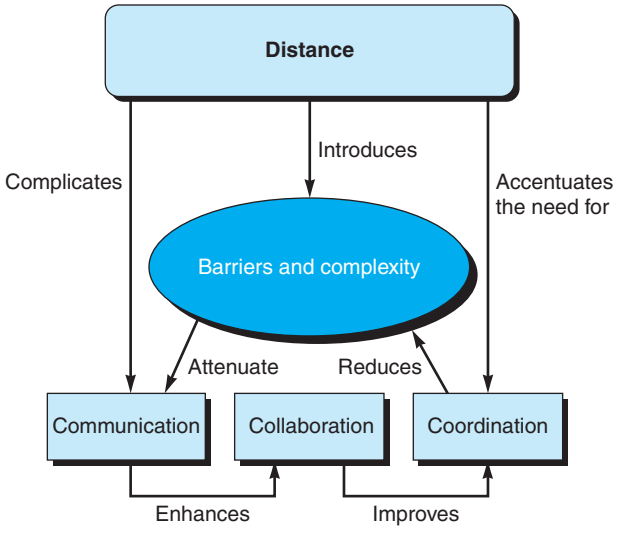
In the software domain, globalization implies more than the transfer of goods and services across international boundaries. For the past few decades, an increasing number of major software products have been built by software teams that are often located in different countries. These global software development (GSD) teams have many of the characteristics of a conventional software team (Section 6.4), but a GSD team has other unique challenges that include coordination, collaboration, communication, and specialized decision making. Approaches to coordination, collaboration, and communication have been discussed earlier in this chapter. Decision making on all software teams is complicated by four factors [Gar10]:

- Complexity of the problem.
- Uncertainty and risk associated with the decision.
- The law of unintended consequences (i.e., work-associated decision has an unintended effect on another project objective).
- Different views of the problem that lead to different conclusions about the way forward.

7 Tools noted here do not represent an endorsement, but rather, a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

FIGURE 6.2

Factors affecting a GSD team (adapted from [Cas06])



For a GSD team, the challenges associated with coordination, collaboration, and communication can have a profound effect on decision making. Figure 6.2 illustrates the impact of distance on the challenges that face a GSD team. Distance complicates communication, but at the same time, accentuates the need for coordination. Distance also introduces barriers and complexity that can be driven by cultural differences. Barriers and complexity attenuate communication (i.e., the signal-to-noise ratio decreases). The problems inherent in this dynamic can result in a project that becomes unstable.

Although there is no silver bullet that can fully correct the relationships implied by Figure 6.2, the use of effective CDEs (Section 6.6) can help reduce the impact of distance.

6.10 SUMMARY

A successful software engineer must have technical skills. But in addition, he must take responsibility for his commitments, be aware of the needs of his peers, be honest in his assessment of the product and the project, be resilient under pressure, treat his peers fairly, and exhibit attention to detail.

The psychology of software engineering includes individual cognition and motivation, the group dynamics of a software team, and the organization behavior of the company. In order to improve communication and collaboration, members of a software team can take on boundary-spanning roles.

A successful ("jelled") software team is more productive and motivated than average. To be effective, a software team must have a sense of purpose, a sense of involvement, a sense of trust, and a sense of improvement. In addition the team

must avoid "toxicity" that is characterized by a frenzied and frustrating work atmosphere, an inappropriate software process, an unclear definition of roles on the software team, and continuous exposure to failure.

There are many different team structures. Some teams organize hierarchically, while others prefer a loose structure that relies on individual initiative. Agile teams subscribe to the agile philosophy and generally have more autonomy than more conventional software teams. Agile teams emphasize communication, simplicity, feedback, courage, and respect.

Social media is becoming an integral part of many software projects. Blogs, microblogs, forums, and social networking capabilities help to form a software engineering community that communicates and coordinates more effectively.

Cloud computing has the potential to influence the manner in which software engineers organize their teams, the way they do their work, the manner in which they communicate and connect, and the way software projects are managed. In situations in which the cloud can enhance the social and collaborative aspects of software development, its benefits far outweigh its risks.

Collaborative development environments contain a number of services that enhance communication and collaboration for a software team. These environments are particularly useful for global software development where geographic separation can precipitate barriers to successful software engineering.

Problems and Points to Ponder

6.1. Based on your personal observation of people who are excellent software developers, name three personality traits that appear to be common among them.

6.2. How can you be "brutally honest" and still not be perceived (by others) as insulting or aggressive?

6.3. How does a software team construct "artificial boundaries" that reduce their ability to communicate with others?

6.4. Write a brief scenario that describes each of the "boundary-spanning roles" described in Section 6.2.

6.5. In Section 6.3, we note that a sense of purpose, involvement, trust, and improvement are essential attributes for effective software teams. Who is responsible for instilling these attributes as a team is formed?

6.6. Which of the four organizational paradigms for teams (Section 6.4) do you think would be most effective (a) for the IT department at a major insurance company; (b) for a software engineering group at a major defense contractor; (c) for a software group that builds computer games; (d) for a major software company? Explain why you made the choices you did.

6.7. If you had to pick one attribute of an agile team that makes it different from a conventional software team, what would it be?

6.8. Of the forms of social media that were described for software engineering work in Section 6.6, which do you think would be most effective and why?

6.9. Write a scenario in which the *SafeHome* team members make use of one or more forms of social media as part of their software project.

6.10. Presently, the cloud is one of the more hyped concepts in the world of computing. Describe how the cloud can add value for a software engineering organization with specific reference to services that are specifically designed to enhance software engineering work.

6.11. Do some research on one of the CDE tools noted in the sidebar in Section 6.8 (or a tool assigned by your instructor) and prepare a brief presentation of the tool's capabilities for your class.

6.12. Referring to Figure 6.2, why does distance complicate communication? Why does distance accentuate the need for coordination? Why types of barriers and complexity are introduced by distance?

Further Readings and Information Sources

Although many books have addressed the human aspects of software engineering, two books can legitimately be called classics. Jerry Weinberg (*The Psychology of Computer Programming,* Silver Anniversary Edition, Dorset House, 1998) was the first to consider the psychology of the people who build computer software. Tom DeMarco and Tim Lister (*Peopleware: Productive Projects and Teams,* 2nd ed., Dorset House, 1999) argue that the major challenges in software development are human, not technical.

Useful insights into the human aspects of software engineering have also been provided by Mantle and Lichty (*Managing the Unmanageable: Rules, Tools, and Insights for Managing Software People and Teams*, Addison-Wesley, 2012), Fowler (*The Passionate Programmer,* Pragmatic Bookshelf, 2009), McConnell (*Code Complete,* 2nd ed., Microsoft Press, 2004), Brooks (*The Mythical Man-Month,* 2nd ed., Addison-Wesley, 1999), and Hunt and Thomas (*The Pragmatic Programmer,* Addison-Wesley, 1999). Tomayko and Hazzan *(Human Aspects of Software Engineering,* Charles River Media, 2004) address both the psychology and sociology of software engineering with an emphasis on XP.

The human aspects of the agile development have been addressed by Rasmussen (*The Agile Samurai,* Pragmatic Bookshelf, 2010) and Davies (*Agile Coaching,* Pragmatic Bookshelf, 2010). Important aspects of agile teams are considered by Adkins (*Coaching Agile Teams,* Addison-Wesley, 2010), and Derby, Larsen, and Schwaber (*Agile Retrospectives: Making Good Teams Great,* Pragmatic Bookshelf, 2006).

Problem solving is a uniquely human activity and is addressed in books by Adair (*Decision Making and Problem Solving Strategies,* Kogan Page, 2010), Roam (*Unfolding the Napkin,* Portfolio Trade, 2009), and Wananabe (*Problem Solving 101,* Portfolio Hardcover, 2009).

Guidelines for facilitating collaboration within a software team are presented by Tabaka (*Collaboration Explained,* Addison-Wesley, 2006). Rosen (*The Culture of Collaboration,* Red Ape Publishing, 2009), Hansen (*Collaboration*, Harvard Business School Press, 2009), and Sawyer (*Group Genius: The Creative Power of Collaboration,* Basic Books, 2007) present strategies and practical guidelines for improving collaboration on technical teams.

Fostering human innovation is the subject of books by Gray, Brown, and Macanufo (*Game Storming,* O-Reilly Media, 2010), Duggan (*Strategic Intuition,* Columbia University Press, 2007), and Hohmann (*Innovation Games,* Addison-Wesley, 2006).

An overall look at global software development is presented by Ebert *(Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing,* Wiley-IEEE Computer Society Press, 2011). Mite and his colleagues (*Agility Across Time and Space: Implementing Agile Methods in Global Software Projects,* Springer, 2010) have edited an anthology that addresses the use of agile teams in global development.

A wide variety of information sources that discuss the human aspects of software engineering are available on the Internet. An up-to-date list of World Wide Web references that are relevant to the software process can be found at the SEPA website: **www.mhhe.com/pressman**.

PART

Two

Modeling

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the principles, concepts, and methods that are used to create high-quality requirements and design models. These questions are addressed in the chapters that follow:

- What concepts and principles guide software engineering practice?
- What is requirements engineering and what are the underlying concepts that lead to good requirements analysis?
- How is the requirements model created and what are its elements?
- What are the elements of a good design?
- How does architectural design establish a framework for all other design actions and what models are used?
- How do we design high-quality software components?
- What concepts, models, and methods are applied as a user interface is designed?
- What is pattern-based design?
- What specialized strategies and methods are used to design WebApps?
- What specialized strategies and methods are used to design mobile apps?

Once these questions are answered you'll be better prepared to apply software engineering practice.

CHAPTER

7 Principles That Guide Practice

Key Concepts

coding principles . . 122
communication principles 110
core principles. . . . 106
deployment principles 125
design modeling principles 117
living modeling principles 120
modeling principles 114
planning principles 112
practice 105
process. 106
requirements modeling principles 116
testing principles. . 123

In a book that explores the lives and thoughts of software engineers, Ellen Ullman [Ull97] depicts a slice of life as she relates the thoughts of a practitioner under pressure:

> I have no idea what time it is. There are no windows in this office and no clock, only the blinking red LED display of a microwave, which flashes 12:00, 12:00, 12:00, 12:00. Joel and I have been programming for days. We have a bug, a stubborn demon of a bug. So the red pulse no-time feels right, like a read-out of our brains, which have somehow synchronized themselves at the same blink rate . . .
>
> What are we working on? . . . The details escape me just now. We may be helping poor sick people or tuning a set of low-level routines to verify bits on a distributed database protocol—I don't care. I should care; in another part of my being—later, perhaps when we emerge from this room full of computers—I will care very much why and for whom and for what purpose I am writing software. But just now: no. I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer . . .

Quick Look

What is it? Software engineering practice is a broad array of principles, concepts, methods, and tools that you must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted.

Who does it? Practitioners (software engineers) and their managers conduct a variety of software engineering tasks.

Why is it important? The software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Practice provides you with the detail you'll need to drive along the road. It tells you where the bridges, the roadblocks, and the forks are located. It helps you understand the concepts and principles that must be understood and followed to drive safely and rapidly. It instructs you on how to drive, where to slow down, and where to speed up. In the context of software engineering, practice is what you do day in and day out as software evolves from an idea to a reality.

What are the steps? Three elements of practice apply regardless of the process model that is chosen. They are: principles, concepts, and methods. A fourth element of practice—tools—supports the application of methods.

What is the work product? Practice encompasses the technical activities that produce all work products that are defined by the software process model that has been chosen.

How do I ensure that I've done it right? First, have a firm understanding of the principles that apply to the work (e.g., design) that you're doing at the moment. Then, be certain that you've chosen an appropriate method for the work, be sure that you understand how to apply the method, use automated tools when they're appropriate for the task, and be adamant about the need for techniques to ensure the quality of work products that are produced.

A dark image of software engineering practice to be sure, but upon reflection, many of the readers of this book will be able to relate to it.

People who create computer software practice the art or craft or discipline[1] that is software engineering. But what is software engineering "practice"? In a generic sense, *practice* is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how-to's to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

Various aspects of software engineering practice will be examined throughout the remainder of this book. In this chapter, our focus is on principles and concepts that guide software engineering practice in general.

7.1 Software Engineering Knowledge

In an editorial published in *IEEE Software*, Steve McConnell [McC99] made the following comment:

> Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, html, C++, Linux, Windows NT, and so on. Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C++, you have to know something about C++ to get your program to work.
>
> You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

McConnell goes on to argue that the body of software engineering knowledge (circa the year 2000) had evolved to a "stable core" that he estimated represented about "75 percent of the knowledge needed to develop a complex system." But what resides within this stable core?

Over the intervening years, we have seen the evolution of new operating systems like iOS or Android and languages like Java, Python, and C#.

1 Some writers argue for one of these terms to the exclusion of the others. In reality, software engineering is all three.

But, as McConnell indicates, core principles—the elemental ideas that guide software engineers in the work that they do—still provide a foundation from which software engineering models, methods, and tools can be applied and evaluated.

7.2 Core Principles

Quote:

"In theory there is no difference between theory and practice. But, in practice, there is."

Jan van de Snepscheut

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

In Chapter 2, we identified a set of general principles that span software engineering process and practice: (1) provide value to end users, (2) keep it simple, (3) maintain the vision (of the product and the project), (4) recognize that others consume (and must understand) what you produce, (5) be open to the future, (6) plan ahead for reuse, and (7) think! Although these general principles are important, they are characterized at such a high level of abstraction that they are sometimes difficult to translate into day-to-day software engineering practice. In the subsections that follow, we take a more detailed look at the core principles that guide process and practice.

7.2.1 Principles That Guide Process

In Part 1 of this book we discussed the importance of the software process and described the many different process models that have been proposed for software engineering work. Regardless of whether a model is linear or iterative, prescriptive or agile, it can be characterized using the generic process framework that is applicable for all process models. The following set of core principles can be applied to the framework, and by extension, to every software process.

ADVICE

Every project and every team is unique. That means that you must adapt your process to best fit your needs.

Principle 1. ***Be agile.*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions locally whenever possible.

Principle 2. ***Focus on quality at every step.*** The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle 3. ***Be ready to adapt.*** Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

Principle 4. ***Build an effective team.*** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.[2]

Principle 5. ***Establish mechanisms for communication and coordination.*** Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.

Quote:

"The truth of the matter is that you always know the right thing to do. The hard part is doing it."

General H. Norman Schwarzkopf

Principle 6. ***Manage change.*** The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

Principle 7. ***Assess risk.*** Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans. Some of these contingency plans will form the basis for security engineering tasks (Chapter 27).

Principle 8. ***Create work products that provide value for others.*** Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design, the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

Part 4 of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

7.2.2 Principles That Guide Practice

Software engineering practice has a single overriding goal—to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders. To achieve this goal, you should adopt a set of core principles that guide your technical work. These principles have merit regardless of the analysis and design methods that you apply, the construction techniques (e.g., programming language, automated tools) that you use, or the

2 The characteristics of effective software teams have been discussed in Chapter 6.

verification and validation approach that you choose. The following set of core principles are fundamental to the practice of software engineering:

Principle 1. ***Divide and conquer.*** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoCs). A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.

Principle 2. ***Understand the use of abstraction.*** At its core, an abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. When we use the abstraction *spreadsheet*, it is assumed that you understand what a spreadsheet is, the general structure of content that a spreadsheet presents, and the typical functions that can be applied to it. In software engineering practice, you use many different levels of abstraction, each imparting or implying meaning that must be communicated. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a *column* or the *SUM* function).

Joel Spolsky [Spo02] suggests that "all non-trivial abstractions, to some degree, are leaky." The intent of an abstraction is to eliminate the need to communicate details. But sometimes, problematic effects precipitated by these details "leak" through. Without an understanding of the details, the cause of a problem cannot be easily diagnosed.

Principle 3. ***Strive for consistency.*** Whether it's creating an analysis model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.

Principle 4. ***Focus on the transfer of information.*** Software is about information transfer—from a database to an end user, from a legacy system to a WebApp, from an end user into a graphic user interface (GUI), from an operating system to an application, from one software component to another—the list is almost endless. In every case, information flows across an interface, and as a consequence, there are opportunities for error, or omission, or ambiguity. The implication of this principle is that you must pay special attention to the analysis, design, construction, and testing of interfaces.

Principle 5. ***Build software that exhibits effective modularity.*** Separation of concerns (Principle 1) establishes a philosophy for software. *Modularity* provides a mechanism for realizing the philosophy. Any complex system

can be divided into modules (components), but good software engineering practice demands more. Modularity must be *effective*. That is, each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents. Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

ADVICE

Use patterns (Chapter 16) to capture knowledge and experience for future generations of software engineers.

Principle 6. ***Look for patterns.*** Brad Appleton [App00] suggests that:

> The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users.

The use of design patterns can be applied to wider systems engineering and systems integration problems, by allowing components in complex systems to evolve independently.

ADVICE

Avoid tunnel vision by examining a problem from a number of different perspectives. You discover aspects that would haven been hidden otherwise.

Principle 7. ***When possible, represent the problem and its solution from a number of different perspectives.*** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered. For example, a requirements model can be represented using a scenario-oriented viewpoint, a class-oriented viewpoint, or a behavioral viewpoint (Chapters 9 through 11). Each provides a different perspective of the problem and its requirements.

Principle 8. ***Remember that someone will maintain the software.*** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

These principles are not all you'll need to build high-quality software, but they do establish a foundation for every software engineering method discussed in this book.

7.3 Principles That Guide Each Framework Activity

In the sections that follow we consider principles that have a strong bearing on the success of each generic framework activity defined as part of the software process. In many cases, the principles that are discussed for each of the

framework activities are a refinement of the principles presented in Section 7.2. They are simply core principles stated at a lower level of abstraction.

7.3.1 Communication Principles

Before customer requirements can be analyzed, modeled, or specified they must be gathered through the communication activity. A customer has a problem that may be amenable to a computer-based solution. You respond to the customer's request for help. Communication has begun. But the road from communication to understanding is often full of potholes.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that you will confront. In this context, we discuss communication principles as they apply to customer communication. However, many of the principles apply equally to all forms of communication that occur within a software project.

ADVICE

Before communicating be sure you understand the point of view of the other party, know a bit about his or her needs, and then listen.

Principle 1. ***Listen.*** Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. *Never* become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2. ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3. ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure that other principles are followed.

Quote:

"Plain questions and plain answers make the shortest road to most perplexities."

Mark Twain

Principle 4. ***Face-to-face communication is best.*** But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

Principle 5. ***Take notes and document decisions.*** Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle 6. ***Strive for collaboration.*** Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle 7. ***Stay focused; modularize your discussion.*** The more people are involved in any communication, the more likely that discussion will

bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved (however, see Principle 9).

Principle 8. ***If something is unclear, draw a picture.*** Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

? What happens if I can't come to an agreement with the customer on some project-related issue?

Principle 9. ***(a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on.*** Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle 10. ***Negotiation is not a contest or a game. It works best when both parties win.*** There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

INFO

The Difference Between Customers and End Users

Software engineers communicate with many different stakeholders, but customers and end users have the most significant impact on the technical work that follows. In some cases the customer and the end user are one and the same, but for many projects, the customer and the end user are different people, working for different managers in different business organizations.

A *customer* is the person or group who (1) originally requested the software to be built, (2) defines overall business objectives for the software, (3) provides basic product requirements, and (4) coordinates funding for the project. In a product or system business, the customer is often the marketing department. In an information technology (IT) environment, the customer might be a business component or department.

An *end user* is the person or group who (1) will actually use the software that is built to achieve some business purpose and (2) will define operational details of the software so the business purpose can be achieved.

SAFEHOME

Communication Mistakes

The scene: Software engineering team workspace

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member.

The conversation:

Ed: What have you heard about this *SafeHome* project?

Vinod: The kick-off meeting is scheduled for next week.

Jamie: I've already done a little bit of investigation, but it didn't go well.

Ed: "What do you mean?"

Jamie: Well, I gave Lisa Perez a call. She's the marketing honcho on this thing.

Vinod: And . . . ?

Jamie: I wanted her to tell me about *SafeHome* features and functions . . . that sort of thing. Instead, she began asking me questions about security systems, surveillance systems . . . I'm no expert.

Vinod: What does that tell you?

(Jamie shrugs)

Vinod: That marketing will need us to act as consultants and that we'd better do some homework on this product area before our kick-off meeting. Doug said that he wanted us to "collaborate" with our customer, so we'd better learn how to do that.

Ed: Probably would have been better to stop by her office. Phone calls just don't work as well for this sort of thing.

Jamie: You're both right. We've got to get our act together or our early communications will be a struggle.

Vinod: I saw Doug reading a book on "requirements engineering." I'll bet that lists some principles of good communication. I'm going to borrow it from him.

Jamie: Good idea . . . then you can teach us.

Vinod (smiling): Yeah, right.

7.3.2 Planning Principles

Quote:

"In preparing for battle I have always found that plans are useless, but planning is indispensable."

General Dwight D. Eisenhower

The communication activity helps you to define your overall goals and objectives (subject, of course, to change as time passes). However, understanding these goals and objectives is not the same as defining a plan for getting there. The planning activity encompasses a set of management and technical practices that enable the software team to define a road map as it travels toward its strategic goal and tactical objectives.

Try as we might, it's impossible to predict exactly how a software project will evolve. There is no easy way to determine what unforeseen technical problems will be encountered, what important information will remain undiscovered until late in the project, what misunderstandings will occur, or what business issues will change. And yet, a good software team must plan its approach.

There are many different planning philosophies.[3] Some people are "minimalists," arguing that change often obviates the need for a detailed plan. Others are "traditionalists," arguing that the plan provides an effective road map and the more detail it has, the less likely the team will become lost. Still others are "agilists," arguing that a quick "planning game" may be necessary, but that the road map will emerge as "real work" on the software begins.

WebRef

An excellent repository of planning and project management information can be found at **www.4pm.com/repository.htm.**

What to do? On many projects, overplanning is time consuming and fruitless (too many things change), but underplanning is a recipe for chaos. Like most things in life, planning should be conducted in moderation, enough to provide

3 A detailed discussion of software project planning and management is presented in Part 4 of this book.

useful guidance for the team—no more, no less. Regardless of the rigor with which planning is conducted, the following principles always apply:

Principle 1. ***Understand the scope of the project.*** It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

Principle 2. ***Involve stakeholders in the planning activity.*** Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

Principle 3. ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate re-planning after the delivery of each software increment based on feedback received from users.

Principle 4. ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle 5. ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur. Take into account the likely exposure due to losses or compromises of project assets.

Quote:

"Success is more a function of consistent common sense than it is of genius."

An Wang

Principle 6. ***Be realistic.*** People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

Principle 7. ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "low-granularity" plan provides broader work tasks that are planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require high granularity (too much can change).

KEY POINT

The term *granularity* refers to the detail with which some element of planning is represented or conducted.

Principle 8. ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality. If technical reviews[4] are to be conducted, they should be scheduled. If pair programming (Chapter 5) is to be used during construction, it should be explicitly defined within the plan.

Principle 9. ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle 10. ***Track the plan frequently and make adjustments as required.*** Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

To be most effective, everyone on the software team should participate in the planning activity. Only then will team members "sign up" to the plan.

7.3.3 Modeling Principles

We create models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (e.g., a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the architecture and functions that enable the transformation to occur, the features that users desire, and the behavior of the system as the transformation is taking place. Models must accomplish these objectives at different levels of abstraction—first depicting the software from the customer's viewpoint and later representing the software at a more technical level.

KEY POINT

Analysis models represent customer requirements. Design models provide a concrete specification for the construction of the software.

In software engineering work, two classes of models can be created: requirements models and design models. *Requirements models* (also called *analysis models*) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain. *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

4 Technical reviews are discussed in Chapter 20.

In their book on agile modeling, Scott Ambler and Ron Jeffries [Amb02b] define a set of modeling principles[5] that are intended for those who use the agile process model (Chapter 5) but are appropriate for all software engineers who perform modeling action and tasks:

Principle 1. ***The primary goal of the software team is to build software, not create models.*** Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

Principle 2. ***Travel light—don't create more models than you need.*** Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that make it easier and faster to construct the software.

ADVICE

The intent of any model is to communicate information. To accomplish this, use a consistent format. Assume that you won't be there to explain the model. It should stand on its own.

Principle 3. ***Strive to produce the simplest model that will describe the problem or the software.*** Don't overbuild the software [Amb02b]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. ***Build models in a way that makes them amenable to change.*** Assume that your models will change, but in making this assumption don't get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.

Principle 5. ***Be able to state an explicit purpose for each model that is created.*** Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

Principle 6. ***Adapt the models you develop to the system at hand.*** It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

5 The principles noted in this section have been abbreviated and rephrased for the purposes of this book.

Principle 7. ***Try to build useful models, but forget about building perfect models.*** When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth the benefits of these properties. Are we suggesting that modeling should be sloppy or low quality? The answer is no. But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.

Principle 8. ***Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.*** Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

Principle 9. ***If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.*** If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

Principle 10. ***Get feedback as soon as you can.*** Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

Requirements modeling principles. Over the past three decades, a large number of requirements modeling methods have been developed. Investigators have identified requirements analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

Quote:

"The engineer's first problem in any design situation is to discover what the problem really is."

Author unknown

Principle 1. ***The information domain of a problem must be represented and understood.*** The *information domain* encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

Principle 2. ***The functions that the software performs must be defined.*** Software functions provide direct benefit to end users and also provide

KEY POINT

Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered, and behavior to be exhibited.

internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Principle 3. ***The behavior of the software (as a consequence of external events) must be represented.*** The behavior of computer software is driven by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.*** Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called *partitioning* or *separation of concerns,* and it is a key strategy in requirements modeling.

Principle 5. ***The analysis task should move from essential information toward implementation detail.*** Analysis modeling begins by describing the problem from the end user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively, a keyboard command might be typed, a game pad joystick (or mouse) might be pointed in a specific direction, a motion-sensitive device might be waved in the air, or a device that reads the player's body movements directly can be used.

By applying these principles, a software engineer approaches a problem systematically. But how are these principles applied in practice? This question will be answered in Chapters 8 through 11.

Design modeling principles. The software design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the

Quote:

"See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect."

William Shakespeare

thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the system.

There is no shortage of methods for deriving the various elements of a software design. Some methods are data driven, allowing the data structure to dictate the program architecture and the resultant processing components. Others are pattern driven, using information about the problem domain (the requirements model) to develop architectural styles and processing patterns. Still others are object oriented, using problem domain objects as the driver for the creation of data structures and the methods that manipulate them. Yet all embrace a set of design principles that can be applied regardless of the method that is used:

WebRef

Insightful comments on the design process, along with a discussion of design aesthetics, can be found at **http://www.gobookee.net/search.php?q=aabyan+design+aesthetics.**

Principle 1. ***Design should be traceable to the requirements model.*** The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

Principle 2. ***Always consider the architecture of the system to be built.*** Software architecture (Chapter 13) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. ***Design of data is as important as design of processing functions.*** Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4. ***Interfaces (both internal and external) must be designed with care.*** The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. ***User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use.*** The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures,

Quote:

"The differences are not minor—they are rather like the differences between Salieri and Mozart. Study after study shows that the very best designers produce structures that are faster, smaller, simpler, clearer, and produced with less effort."

Frederick P. Brooks

no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. ***Component-level design should be functionally independent.*** Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.[6]

Principle 7. ***Components should be loosely coupled to one another and to the external environment.*** Coupling is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling should be kept as low as is reasonable.

Principle 8. ***Design representations (models) should be easily understandable.*** The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. ***The design should be developed iteratively.*** With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors, but later iterations should strive to make the design as simple as is possible.

Principle 10. ***Creation of a design model does not preclude an agile approach.*** Some proponents of agile software development (Chapter 5) insist that the code is the only design documentation that is needed. Yet the purpose of a design model is to help others who must maintain and evolve the system. It is extremely difficult to understand either the higher level purpose of a code fragment or its interactions with other modules in a modern multithreaded run-time environment.

Although in-line code documentation can be useful, it is often difficult to keep code and code descriptions consistent. The design model provides benefit because it is created at a level of abstraction that is stripped of unnecessary technical detail and is closely coupled to the application concepts and requirements.

Complementary design information can incorporate a design rationale including the descriptions of rejected architectural design alternatives. This information may be needed to help you see through the code forest. In addition, it can help maintain consistency when finer-grained design decisions are required.

6 Additional discussion of cohesion can be found in Chapter 12.

This type of architectural specification can also help diverse system stakeholders communicate with the design team and each other.

With the exception of relatively small systems that can be prototyped and experimented with quickly, doing high-level design using only source code is unwise. Agile design documentation keeps step with design and development. To avoid waste, the effort expended on these documents should be proportional to the stability of the design. In the early stages of design, descriptions must be adequate to communicate with stakeholders. The more stable the design the more extensive the descriptions. One approach might be to use design modeling tools that produce executable models that can be evaluated in the usual agile manner.

When these design principles are properly applied, you create a design that exhibits both external and internal quality factors [Mye78]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts (Chapter 12).

Living modeling principles. Breu [Bre10] describes *living models* as a paradigm that combines model-based development[7] with the management and operation of service-oriented systems.[8] Living models support cooperation among all project stakeholders by providing appropriate model-based abstractions that describe interdependencies among system elements. There are eight principles that are crucial for establishing a living models environment:

> **Principle 1.** ***Stakeholder-centric models should target specific stakeholders and their tasks.*** This means that stakeholders are allowed to operate on the models at a level of abstraction that is appropriate, and that lower levels are hidden from them. For example, the CIO is concerned with business processes while a tester needs to formulate test cases at the requirements level.
>
> **Principle 2.** ***Models and code should be closely coupled.*** If an operable system is the main target, any model that does not reflect the operable system is useless. This means that the code and model need to be in consistent states. Tools can be used to support linking models and the code.
>
> **Principle 3.** ***Bidirectional information flow should be established between models and code.*** Changes within the model, code, and operable system must be allowed to propagate when they occur. Traditionally, changes

7 *Model-based development* (also called *model-driven engineering*) builds domain models that depict specific aspects of an application domain.

8 A *service-oriented system* packages software functionality in the form of services that are accessible through a networked infrastructure.

made at the code level are reflected in the running system. It is also important to have those code changes reflected in the model.

Principle 4. ***A common system view should be created.*** A system meta model defines business processes and information objects in the IT management layer, running services and physical nodes in the systems operations layer, and a requirements view in the software engineering layer. The associations in the system meta model describe dependencies from business processes and business objects to the technology layer.

Principle 5. ***The information in the model must be persistent to allow tracking of system changes.*** The system model describes the current state of the system at all levels of abstraction. System evolution may be described and documented as a sequence of system model snapshots.

Principle 6. ***Information consistency across all levels of the model must be verified.*** Model constraint checking and state information retrieval are two important services required to support stakeholder decision making. For example, a software architect may need to check to see that each service at the requirements level has a corresponding service at the architecture level.

Principle 7. ***Each model element has assigned stakeholder rights and responsibilities.*** Each stakeholder is responsible for an identified subset of model elements. Each model subset is a stakeholder's domain. This means that each model element has access to information describing the actions each stakeholder is able to perform on the element.

Principle 8. ***The states of various model elements should be represented.*** Just as the state of computation is defined by the values held by key variables during run time, the state of each model element can be defined by the values assigned to its attributes.

Quote:

"For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change."

David Parnas

7.3.4 Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. In modern software engineering work, coding may be (1) the direct creation of programming language source code (e.g., Java), (2) the automatic generation of source code using an intermediate designlike representation of the component to be built (e.g., Enterprise Architect),[9] or (3) the automatic generation of executable code using a fourth-generation programming language (e.g., Visual C#).

9 Enterprise Architect is tool created by Sparx Systems http://www.sparxsystems.com/products/ea/index.html

The initial focus of testing is at the component level, often called *unit testing*. Other levels of testing include (1) *integration testing* (conducted as the system is constructed), (2) *validation testing* that assesses whether requirements have been met for the complete system (or software increment), and (3) *acceptance testing* that is conducted by the customer in an effort to exercise all required features and functions. The following set of fundamental principles and concepts are applicable to coding and testing.

Coding principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

ADVICE

Avoid developing an elegant program that solves the wrong problem. Pay particular attention to the first preparation principle.

Preparation Principles: *Before you write one line of code, be sure you*

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles: *As you begin writing code, be sure you*

- Constrain your algorithms by following structured programming [Boh00] practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.
- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles: *After you've completed your first coding pass, be sure you*

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

WebRef

A wide variety of links to coding standards can be found at **http://www.literateprogramming.com/links.html.**

More books have been written about programming (coding) and the principles and concepts that guide it than about any other topic in the software process. Books on the subject include early works on programming style [Ker78], practical software construction [McC04], programming pearls [Ben99], the art of programming [Knu98], pragmatic programming issues [Hun99], and many, many other subjects. A comprehensive discussion of these principles and concepts is beyond the scope of this book. If you have further interest, examine one or more of the references noted.

? What are the objectives of software testing?

Testing principles. In a classic book on software testing, Glen Myers [Mye79] states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint for some software developers. They move counter to the commonly held view that a successful test is one in which no errors are found. Your objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

ADVICE

In a broader software design context, recall that we begin "in the large" by focusing on software architecture and end "in the small" focusing on components. For testing, we simply reverse the focus and test our way out.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, and that behavioral and performance requirements appear to have been met. In addition, the data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects; it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

Davis [Dav95b] suggests a set of testing principles[10] that have been adapted for use in this book. In addition, Everett and Meyer [Eve09] suggest additional principles:

> **Principle 1.** ***All tests should be traceable to customer requirements.***[11] The objective of software testing is to uncover errors. It follows that the most

10 Only a small subset of Davis's testing principles are noted here. For more information, see [Dav95b].

11 This principle refers to *functional tests*, that is, tests that focus on requirements. *Structural tests* (tests that focus on architectural or logical detail) may not address specific requirements directly.

severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. ***Tests should be planned long before testing begins.*** Test planning (Chapter 22) can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

Principle 3. ***The Pareto principle applies to software testing.*** In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. ***Testing should begin "in the small" and progress toward testing "in the large."*** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Principle 5. ***Exhaustive testing is not possible.*** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Principle 6. ***Apply to each module in the system a testing effort commensurate with its expected fault density.*** These are often the newest modules or the ones that are least understood by the developers.

Principle 7. ***Static testing techniques can yield high results.*** More than 85% of software defects originated in the software documentation (requirements, specifications, code walkthroughs, and user manuals) [Jon91]. There may be value in testing the system documentation.

Principle 8. ***Track defects and look for patterns in defects uncovered by testing.*** The total defects uncovered is a good indicator of software quality. The types of defects uncovered can be a good measure of software stability. Patterns of defects found over time can forecast numbers of expected defects.

Principle 9. ***Include test cases that demonstrate software is behaving correctly.*** As software components are being maintained or adapted, unexpected interactions cause unintended side effects in other components. It is important to have a set of regression test cases (Chapter 22) ready to check system behavior after changes are applied to a software product.

7.3.5 Deployment Principles

As we noted in Part 1 of this book, the deployment activity encompasses three actions: delivery, support, and feedback. Because modern software process models are evolutionary or incremental in nature, deployment happens not once, but a number of times as software moves toward completion. Each delivery cycle provides the customer and end users with an operational software increment that provides usable functions and features. Each support cycle provides documentation and human assistance for all functions and features introduced during all deployment cycles to date. Each feedback cycle provides the software team with important guidance that results in modifications to the functions, features, and approach taken for the next increment.

The delivery of a software increment represents an important milestone for any software project. A number of key principles should be followed as the team prepares to deliver an increment:

ADVICE

Be sure that your customer knows what to expect before a software increment is delivered. Otherwise, you can bet the customer will expect more than you deliver.

Principle 1. ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [Kar94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2. ***A complete delivery package should be assembled and tested.*** All executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in all possible computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements).

Principle 3. ***A support regime must be established before the software is delivered.*** An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate record-keeping mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle 4. ***Appropriate instructional materials must be provided to end users.*** The software team delivers more than the software itself.

Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.[12]

Principle 5. ***Buggy software should be fixed first, delivered later.*** Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

The delivered software provides benefit for the end user, but it also provides useful feedback for the software team. As the increment is put into use, end users should be encouraged to comment on features and functions, ease of use, reliability, security concerns, and any other characteristics that are appropriate.

7.4 Work Practices

Quote:

"The ideal engineer is a composite . . . He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems."

N. W. Dougherty

Iskold [Isk08] writes that the quality of software has become the competitive differentiator between software companies. As you learned in Chapter 6, the human aspects of software engineering are as important as any other technology area. For that reason, it is interesting to examine the traits and work habits that seem to be shared among successful software engineers. Among the more important are a desire to continuously refactor the design and code, actively use proven design patterns, acquire reusable components whenever possible, focus on usability, develop maintainable applications, apply the programming language that is best for the application, and build software using proven design and testing practices.

Beyond basic traits and work habits, Isklod [Isk08] suggests 10 concepts that transcend programming languages and specific technologies. Some of these concepts form the prerequisite knowledge needed to appreciate the role of software engineering in the software process.

1. *Interfaces.* Simple, familiar interfaces are less error-prone than complex or unique interfaces.
2. *Conventions and templates.* Naming conventions and software templates are a good way to communicate with a larger number of developers and end users.

12 During the communication activity, the software team should determine what types of help materials users want.

3. *Layering*. Layering is the key to both data and programming abstractions. It allows a separation of design concepts and implementation details and, at the same time, reduces the complexity of the software design.
4. *Algorithmic complexity*. Software engineers must be able to appreciate the elegance and performance characteristics of algorithms, even when selecting among library routines. Writing simple and readable code is often a good way to ensure the time and space efficiency of an application.
5. *Hashing*. Hashes are important for efficient storage and retrieval of data. Hashes can also be important as a means to allocate data evenly among computers in a cloud database.
6. *Caching*. Software engineers need to appreciate the trade-offs associated with providing quick access to a subset of data by storing it in computer memory and not secondary storage devices. Thrashing may occur when mutually dependent data are not in memory at the same time. Applications can slow down when new information needs to be brought into memory (e.g., playing cut scenes in a real-time video game).
7. *Concurrency*. The widespread availability of multiprocessor computers and multithreaded programming environments creates software engineering challenges.
8. *Cloud computing*. Cloud computing provides powerful and readily accessible web services and data to computing platforms of all types.
9. *Security*. Protecting the confidentiality and integrity of system assets should be the concern of every computing professional.
10. *Relational databases*. Relational databases are the cornerstone of information storage and retrieval. It is important to know how to minimize data redundancy and to maximize the speed of retrieval.

In many cases a few good software engineers working "smart" can be more productive than groups many times their size. A good software engineer must know what principles, practices, and tools to use, when to use them, and why they are needed.

7.5 Summary

Software engineering practice encompasses principles, concepts, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different. Yet, a set of generic principles apply to the process as a whole and to the practice of each framework activity regardless of the project or the product.

A set of core principles help in the application of a meaningful software process and the execution of effective software engineering methods. At the process

level, core principles establish a philosophical foundation that guides a software team as it navigates through the software process. At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles provide guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modeling encompasses both analysis and design, describing representations of the software that progressively become more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software. Modeling principles serve as a foundation for the methods and notation that are used to create representations of the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested. Coding principles define generic actions that should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: testing is a process of executing a program with the intent of finding an error.

Deployment occurs as each software increment is presented to the customer and encompasses delivery, support, and feedback. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support information for the software. Support demands advance preparation. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.

Problems and Points to Ponder

7.1. Since a focus on quality demands resources and time, is it possible to be agile and still maintain a quality focus?

7.2. Of the eight core principles that guide process (discussed in Section 7.2.1), which do you believe is most important?

7.3. Describe the concept of *separation of concerns* in your own words.

7.4. An important communication principle states, "Prepare before you communicate." How should this preparation manifest itself in the early work that you do? What work products might result as a consequence of early preparation?

7.5. Do some research on "facilitation" for the communication activity (use the references provided or others) and prepare a set of guidelines that focus solely on facilitation.

7.6. How does agile communication differ from traditional software engineering communication? How is it similar?

7.7. Why is it necessary to "move on"?

7.8. Do some research on "negotiation" for the communication activity and prepare a set of guidelines that focus solely on negotiation.

7.9. Describe what *granularity* means in the context of a project schedule.

7.10. Why are models important in software engineering work? Are they always necessary? Are there qualifiers to your answer about necessity?

7.11. What three "domains" are considered during requirements modeling?

7.12. Try to add one additional principle to those stated for coding in Section 7.3.4.

7.13. What is a successful test?

7.14. Do you agree or disagree with the following statement: "Since we deliver multiple increments to the customer, why should we be concerned about quality in the early increments—we can fix problems in later iterations." Explain your answer.

7.15. Why is feedback important to the software team?

Further Readings and Information Sources

Customer communication is a critically important activity in software engineering, yet few practitioners spend any time reading about it. Withall (*Software Requirements Patterns,* Microsoft Press, 2007) presents a variety of useful patterns that address communications problems. van Lamsweerde (*Requirement Engineering: From System Goals to UML Models to Software Specifications,* Wiley, 2009) and Sutliff (*User-Centered Requirements Engineering,* Springer, 2002) focuses heavily on communications-related challenges.

Books by Karten (*Changing How You Manage and Communicate Change,* IT Governace Publishing, 2009), Weigers (*Software Requirements,* 2nd ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer,* Dorset House, 1996), and Karten [Kar94] provide much insight into methods for effective customer interaction. Although their book does not focus on software, Hooks and Farry (*Customer-Centered Products*, American Management Association, 2000) present useful generic guidelines for customer communication. Young (*Project Requirements: A Guide to Best Practices,* Management Concepts, 2006 and *Effective Requirements Practices,* Addison-Wesley, 2001) emphasizes a "joint team" of customers and developers who develop requirements collaboratively. Hull, Jackson, and Dick (*Requirements Engineering,* Springer, 3rd ed., 2010) and Somerville and Kotonya (*Requirements Engineering: Processes and Techniques,* Wiley, 1998) discuss "elicitation" concepts and techniques and other requirements engineering principles.

Communication and planning concepts and principles are considered in many project management books. Useful project management offerings include books by Juli (*Leadership Principles for Project Success,* CRC Press, 2012), West and his colleagues (*Project Management for IT Related Projects,* British Informatics Society, 2012), Wysocki (*Effective Project Management: Agile, Adaptive, Extreme,* 5th ed., Wiley, 2009), Hughes (*Software Project Management,* 5th ed., McGraw-Hill, 2009), Bechtold (*Essentials of Software Project Management,* 2nd ed., Management Concepts, 2007), Leach (*Lean Project Management: Eight Principles for Success,* BookSurge Publishing, 2006), and Stellman and Greene (*Applied Software Project Management,* O'Reilly Media, 2005).

Davis [Dav95b] has compiled an excellent collection of software engineering principles. In addition, virtually every book on software engineering contains a useful discussion of

concepts and principles for analysis, design, and testing. Among the most widely used offerings (in addition to this book!) are:

Abran, A., and J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge,* IEEE, 2002.[13]

Pfleeger, S., *Software Engineering: Theory and Practice,* 4th ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering,* McGraw-Hill, 8th ed., 2010.

Sommerville, I., *Software Engineering,* 9th ed., Addison-Wesley, 2010.

These books also present detailed discussion of modeling and construction principles.

Modeling principles are considered in many books dedicated to requirements analysis and/or software design. Books by Lieberman (*The Art of Software Modeling,* Auerbach, 2007), Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Roques (*UML in Practice,* Wiley, 2004), Penker and Eriksson (*Business Modeling with UML: Business Patterns at Work,* Wiley, 2001) discuss modeling principles and methods.

Norman's (*The Design of Everyday Things,* Basic Books, 2002) is must reading for every software engineer who intends to do design work. Winograd and his colleagues (*Bringing Design to Software,* Addison-Wesley, 1996) have edited an excellent collection of essays that address practical issues for software design. Constantine and Lockwood (*Software for Use,* Addison-Wesley, 1999) present the concepts associated with "user-centered design." Tognazzini (*Tog on Software Design,* Addison-Wesley, 1995) presents a worthwhile philosophical discussion of the nature of design. Stahl and his colleagues (*Model-Driven Software Development: Technology, Engineering,* Wiley, 2006) discuss the principles of model-driven development. Halladay (*Principle-Based Refactoring,* Principle Publishing, 2012) considers eight fundamental design principles and identifies 50 rules for refactoring.

Hundreds of books address one or more elements of the construction activity. Kernighan and Plauger [Ker78] have written a classic text on programming style, McConnell [McC04] presents pragmatic guidelines for practical software construction, Bentley [Ben99] suggests a wide variety of programming pearls, Knuth [Knu98] has written a classic three-volume series on the art of programming, and Hunt [Hun99] suggests pragmatic programming guidelines.

Myers and his colleagues (*The Art of Software Testing,* 3rd ed., Wiley, 2011) have developed a major revision of his classic text and discuss many important testing principles. Books by *How Google Tests Software,* Addison-Wesley, 2012), Perry (*Effective Methods for Software Testing,* 3rd ed., Wiley, 2006), and Whittaker (*How to Break Software,* Addison-Wesley, 2002), Kaner and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001), and Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) each present important testing concepts and principles and much pragmatic guidance.

A wide variety of information sources on software engineering practice are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software engineering practice can be found at the SEPA website: **www.mhhe.com/pressman**.

13 Available free of charge at http://www.computer.org/portal/web/swebok/v3guide

CHAPTER

8

Understanding Requirements

Key Concepts

analysis patterns . 157
collaboration. 140
elaboration. 135
elicitation 134
inception. 133
negotiation. 135
negotiation. 159
quality function deployment. 146
requirements engineering. 132
requirements gathering 143
requirements management. 138
requirements monitoring 160
specification 135

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard. After all, doesn't the customer know what is required? Shouldn't the end users have a good understanding of the features and functions that will provide benefit? Surprisingly, in many instances the answer to these questions is "no." And even if customers and end users are explicit in their needs, those needs will change throughout the project.

In the forward to a book by Ralph Young [You01] on effective requirements practices, one of us [RSP] wrote:

> It's your worst nightmare. A customer walks into your office, sits down, looks you straight in the eye, and says, "I know you think you understand what I said, but what you don't understand is what I said is not what I meant." Invariably, this happens late in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

Quick Look

What is it? Before you begin any technical work, it's a good idea to create a set of requirements for any engineering tasks. These tasks lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software.

Who does it? Software engineers (sometimes referred to as system engineers or "analysts" in the IT world) and other project stakeholders (managers, customers, and end users) all participate in requirements engineering.

Why is it important? Designing and building an elegant computer program that solves the wrong problem serves no one's needs. That's why it's important to understand what the customer wants before you begin to design and build a computer-based system.

What are the steps? Requirements engineering begins with inception (a task that defines the scope and nature of the problem to be solved). It moves onward to elicitation (a task that helps stakeholders define what is required), and then elaboration (where basic requirements are refined and modified). As stakeholders define the problem, negotiation occurs (what are the priorities, what is essential, when is it required?) Finally, the problem is specified in some manner and then reviewed or validated to ensure that your understanding of the problem and the stakeholders' understanding of the problem coincide.

What is the work product? The intent of requirements engineering is to provide all parties with a written understanding of the problem. This can be achieved though a number of work products: usage scenarios, functions and features lists, requirements models, or a specification.

How do I ensure that I've done it right? Requirements engineering work products are reviewed with stakeholders to ensure that what you have learned is what they really meant. A word of warning: Even after all parties agree, things will change, and they will continue to change throughout the project.

stakeholders 139
use cases 149
validating requirements 161
validation 136
viewpoints 139
work products . . . 147

> All of us who have worked in the systems and software business for more than a few years have lived this nightmare, and yet, few of us have learned to make it go away. We struggle when we try to elicit requirements from our customers. We have trouble understanding the information that we do acquire. We often record requirements in a disorganized manner, and we spend far too little time verifying what we do record. We allow change to control us, rather than establishing mechanisms to control change. In short, we fail to establish a solid foundation for the system or software. Each of these problems is challenging. When they are combined, the outlook is daunting for even the most experienced managers and practitioners. But solutions do exist.

It's reasonable to argue that the techniques we'll discuss in this chapter are not a true "solution" to the challenges just noted. But they do provide a solid approach for addressing these challenges.

8.1 Requirements Engineering

Quote:

"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

Fred Brooks

KEY POINT

Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs.

Designing and building computer software is challenging, creative, and just plain fun. In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed. They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program, and that all else is secondary. What makes these arguments seductive is that they contain elements of truth.[1] But each argument is flawed and can lead to a failed software project.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *requirements engineering*. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction. But where does the bridge originate? One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, and end users), where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. Others might suggest that it

1 This is particularly true for small projects (less than one month) and smaller, relatively simple software efforts. As software grows in size and complexity, these arguments begin to break down.

begins with a broader system definition, where software is but one component of the larger system domain. But regardless of the starting point, the journey across the bridge takes you high above the project, allowing you to examine the context of the software work to be performed; the specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Over the past decade, there have been many technology changes that impact the requirements engineering process [Wev11]. Ubiquitous computing allows computer technology to be integrated into many everyday objects. When these objects are networked they can allow the creation of more complete user profiles, with the accompanying concerns for privacy and security.

ADVICE

Expect to do a bit of design during requirements work and a bit of requirements work during design.

Widespread availability of applications in the electronic marketplace will lead to more diverse stakeholder requirements. Stakeholders can customize a product to meet specific, targeted requirements that are applicable to only a small subset of all end users. As product development cycles shorten, there are pressures to streamline requirements engineering so that products come to market more quickly. But the fundamental problem remains the same, getting timely, accurate, and stable stakeholder input.

Requirements engineering encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Quote:

"The seeds of major software disasters are usually sown in the first three months of commencing the software project."

Caper Jones

Inception. How does a software project get started? Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time? There are no definitive answers to these questions. In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort. But in general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.[2] At project inception,[3] you establish a basic understanding of the problem,

2 If a computer-based system is to be developed, discussions begin within the context of a system engineering process. For a detailed discussion of system engineering, visit the website that accompanies this book: www.mhhe.com/pressman

3 Recall that the Unified Process (Chapter 4) defines a more comprehensive "inception phase" that encompasses the inception, elicitation, and elaboration tasks discussed in this chapter.

the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

An important part of elicitation is to establish business goals [Cle10]. Your job is to engage stakeholders and to encourage them to share their goals honestly. Once the goals have been captured, a prioritization mechanism should be established, and a design rationale for a potential architecture (that meets stakeholder goals) can be created.

INFO

Goal-Oriented Requirements Engineering

A *goal* is a long-term aim that a system or product must achieve. Goals may deal with either functional or nonfunctional (e.g., reliability, security, usability, etc.) concerns. Goals are often a good way to explain requirements to stakeholders and, once established, can be used to manage conflicts among stakeholders.

Object models (Chapters 10 and 11) and requirements can be derived systematically from goals. A goal graph showing links among goals can provide some degree of traceability (Section 8.2.6) between high-level strategic concerns to low-level technical details. Goals should be specified precisely and serve as the basis for requirements elaboration, verification/validation, conflict management, negotiation, explanation, and evolution.

Conflicts detected in requirements are often a result of conflicts present in the goals themselves. Conflict resolution is achieved by negotiating a set of mutually agreed-upon goals that are consistent with one another and with stakeholder desires. A more complete discussion on goals and requirements engineering can be found in a paper by Lamsweweerde [LaM01b].

? Why is it difficult to gain a clear understanding of what the customer wants?

Christel and Kang [Cri92] identify a number of problems that are encountered as elicitation occurs. *Problems of scope* occur when the boundary of the system is ill-defined or the customers and users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives. *Problems of understanding* are encountered when customers and users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers and users, or specify requirements that are ambiguous or untestable. *Problems of volatility* occur when the requirements change over time. To help

overcome these problems, you must approach the requirements-gathering activity in an organized manner.

ADVICE

Elaboration is a good thing, but you have to know when to stop. The key is to describe the problem in a way that establishes a firm base for design. If you work beyond that point, you're doing design.

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model (Chapters 9 through 11) that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services[4] that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

ADVICE

There should be no winner and no loser in an effective negotiation. Both sides win, because a "deal" that both can live with is solidified.

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification. In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

KEY POINT

The formality and format of a specification varies with the size and the complexity of the software to be built.

Some suggest that a "standard template" [Som97] should be developed and used for a specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

4 A *service* manipulates the data encapsulated by the class. The terms *operation* and *method* are also used. If you are unfamiliar with object-oriented concepts, a basic introduction is presented in Appendix 2.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document Conventions
   - 1.3 Intended Audience and Reading Suggestions
   - 1.4 Project Scope
   - 1.5 References
2. **Overall Description**
   - 2.1 Product Perspective
   - 2.2 Product Features
   - 2.3 User Classes and Characteristics
   - 2.4 Operating Environment
   - 2.5 Design and Implementation Constraints
   - 2.6 User Documentation
   - 2.7 Assumptions and Dependencies
3. **System Features**
   - 3.1 System Feature 1
   - 3.2 System Feature 2 (and so on)
4. **External Interface Requirements**
   - 4.1 User Interfaces
   - 4.2 Hardware Interfaces
   - 4.3 Software Interfaces
   - 4.4 Communications Interfaces
5. **Other Nonfunctional Requirements**
   - 5.1 Performance Requirements
   - 5.2 Safety Requirements
   - 5.3 Security Requirements
   - 5.4 Software Quality Attributes
6. **Other Requirements**

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted in this sidebar.

ADVICE

A key concern during requirements validation is consistency. Use the analysis model to ensure that requirements have been consistently stated.

Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification[5] to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review (Chapter 20). The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when

5 Recall that the nature of the specification will vary with each project. In some cases, the "specification" is a collection of user scenarios and little else. In others, the specification may be a document that contains scenarios, models, and written descriptions.

large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

To illustrate some of the problems that occur during requirements validation, consider two seemingly innocuous requirements:

- The software should be user friendly.
- The probability of a successful unauthorized database intrusion should be less than 0.0001.

The first requirement is too vague for developers to test or assess. What exactly does "user friendly" mean? To validate it, it must be quantified or qualified in some manner.

The second requirement has a quantitative element ("less than 0.0001"), but intrusion testing will be difficult and time consuming. Is this level of security even warranted for the application? Can other complementary requirements associated with security (e.g., password protection, specialized handshaking) replace the quantitative requirement noted?

Glinz [Gli09] writes that quality requirements need to be represented in a manner that delivers optimal value. This means assessing the risk (Chapter 35) of delivering a system that fails to meet the stakeholders' quality requirements and attempting to mitigate this risk at minimum cost. The more critical the quality requirement is, the greater the need to state it in quantifiable terms. Less-critical quality requirements can be stated in general terms. In some cases, a general quality requirement can be verified using a qualitative technique (e.g., user survey or check list). In other situations, quality requirements can be verified using a combination of qualitative and quantitative assessment.

INFO

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.[6] Many of these activities are identical to the software configuration management (SCM) techniques discussed in Chapter 29.

SOFTWARE TOOLS

Requirements Engineering

Objective: Requirements engineering tools assist in requirements gathering, requirements modeling, requirements management, and requirements validation.

Mechanics: Tool mechanics vary. In general, requirements engineering tools build a variety of graphical (e.g., UML) models that depict the informational, functional, and behavioral aspects of a system. These models form the basis for all other activities in the software process.

Representative Tools:[7]
A reasonably comprehensive (and up-to-date) listing of requirements engineering tools can be found at the Volvere Requirements resources site at **www.volere.co.uk/tools.htm**. Requirements modeling tools are discussed in Chapters 9 and 10. Tools noted below focus on requirement management.

EasyRM, developed by Cybernetic Intelligence GmbH (**http://www.visuresolutions.com/visure-requirements-software**), Visure Requirements is a flexible and complete requirements engineering life-cycle solution, supporting requirements capture, analysis, specification, validation and verification, management, and reuse.

Rational RequisitePro, developed by Rational Software (**www-03.ibm.com/software/products/us/en/reqpro**), allows users to build a requirements database; represent relationships among requirements; and organize, prioritize, and trace requirements.

Many additional requirements management tools can be found at the Volvere site noted earlier and at **www.jiludwig.com/Requirements_Management_Tools.html**.

8.2 ESTABLISHING THE GROUNDWORK

In an ideal setting, stakeholders and software engineers work together on the same team.[8] In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have

6 Formal requirements management is initiated only for large projects that have hundreds of identifiable requirements. For small projects, this requirements engineering function is considerably less formal.

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8 This approach is strongly recommended for projects that adopt an agile software development philosophy.

limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation.

In the sections that follow, we discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

8.2.1 Identifying Stakeholders

KEY POINT

A *stakeholder* is anyone who has a direct interest in or benefits from the system that is to be developed.

Sommerville and Sawyer [Som97] define a *stakeholder* as "anyone who benefits in a direct or indirect way from the system which is being developed." We have already identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

At inception, you should create a list of people who will contribute input as requirements are elicited (Section 8.3). The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

8.2.2 Recognizing Multiple Viewpoints

Quote:

"Put three stakeholders in a room and ask them what kind of system they want. You're likely to get four or more different opinions."

Author unknown

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

Each of these constituencies (and others) will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another. You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

There are several things that can make it hard to elicit requirements for software that satisfies its users: project goals are unclear, stakeholders' priorities differ, people have unspoken assumptions, stakeholders interpret meanings differently, and requirements are stated in a way that makes them difficult to

verify [Ale11]. The goal of effective requirements engineering is to eliminate or at least reduce these problems.

8.2.3 Working toward Collaboration

If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements. Throughout earlier chapters, we have noted that customers (and other stakeholders) should collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result. But how is this collaboration accomplished?

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). It is, of course, the latter category that presents a challenge.

INFO

Using "Priority Points"

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points*. All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of each (from his or her viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Quote:

"It is better to know some of the questions than all of the answers."

James Thurber

8.2.4 Asking the First Questions

Questions asked at the inception of the project should be "context free" [Gau89]. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

? What questions will help you gain a preliminary understanding of the problem?

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg [Gau89] call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Quote:

"He who asks a question is a fool for five minutes; he who does not ask a question is a fool forever."

Chinese proverb

These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a requirements elicitation format that combines elements of problem solving, negotiation, and specification. An approach of this type is presented in Section 8.3.

8.2.5 Nonfunctional Requirements

A *nonfunctional requirement* (NFR) can be described as a quality attribute, a performance attribute, a security attribute, or a general constraint on a system. These are often not easy for stakeholders to articulate. Chung [Chu09] suggests that there is a lopsided emphasis on functionality of the software, yet the software may not be useful or usable without the necessary non-functional characteristics.

In Section 8.3.2, we discuss a technique called *quality function deployment* (QFD). Quality function deployment attempts to translate unspoken customer

needs or goals into system requirements. Nonfunctional requirements are often listed separately in a software requirements specification.

As an adjunct to QFD, it is possible to define a two-phase approach [Hne11] that can assist a software team and other stakeholders in identifying nonfunctional requirements. During the first phase, a set of software engineering guidelines is established for the system to be built. These include guidelines for best practice, but also address architectural style (Chapter 13) and the use of design patterns (Chapter 16). A list of NFRs (e.g., requirements that address usability, testability, security or maintainability) is then developed. A simple table lists NFRs as *column labels* and software engineering guidelines as *row labels*. A relationship matrix compares each guideline to all others, helping the team to assess whether each pair of guidelines is *complementary*, *overlapping*, *conflicting*, or *independent*.

In the second phase, the team prioritizes each nonfunctional requirement by creating a homogeneous set of nonfunctional requirements using a set of decision rules [Hne11] that establish which guidelines to implement and which to reject.

8.2.6 Traceability

Traceability is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases). A *traceability matrix* allows a requirements engineer to represent the relationship between requirements and other software engineering work products. Rows of the traceability matrix are labeled using requirement names and columns can be labeled with the name of a software engineering work product (e.g., a design element or a test case). A matrix cell is marked to indicate the presence of a link between the two.

The traceability matrices can support a variety of engineering development activities. They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used. Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

As the number of requirements and the number of work products grows, it becomes increasingly difficult to keep the traceability matrix up to date. Nonetheless, it is important to create some means for tracking the impact and evolution of the product requirements [Got11].

8.3 Eliciting Requirements

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering,

stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements [Zah90].[9]

8.3.1 Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

? What are the basic guidelines for conducting a collaborative requirements gathering meeting?

- Meetings (either real or virtual) are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

WebRef

Joint Application Development (JAD) is a popular technique for requirements gathering. A good description can be found at **www.carolla.com/wp-jad.htm.**

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

A one- or two-page "product request" is generated during inception (Section 8.2). A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

As an example,[10] consider an excerpt from a product request written by a marketing person involved in the *SafeHome* project. This person writes the following narrative about the home security function that is to be part of *SafeHome:*

> Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with "alarm systems" so this would be an easy sell.
>
> The home security function would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, carbon monoxide levels, and others. It'll use our wireless sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

9 This approach is sometimes called a *facilitated application specification technique* (FAST).

10 This example (with extensions and variations) is used to illustrate important software engineering methods in many of the chapters that follow. As an exercise, it would be worthwhile to conduct your own requirements-gathering meeting and develop a set of lists for it.

ADVICE

If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross section of users. If only one user defines all requirements, acceptance risk is high.

In reality, others would contribute to this narrative during the requirements-gathering meeting and considerably more information would be available. But even with additional information, ambiguity is present, omissions are likely to exist, and errors might occur. For now, the preceding "functional description" will suffice.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

Quote:

"Facts do not cease to exist because they are ignored."

Aldous Huxley

Objects described for *SafeHome* might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on. The list of services might include *configuring* the system, *setting* the alarm, *monitoring* the sensors, *dialing* the phone, *programming* the control panel, and *reading* the display (note that services act on objects). In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

ADVICE

Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. Alternatively, the lists may have been posted on a group forum, at an internal website, or posed in a social networking environment for review prior to the meeting. Ideally, each listed entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted, and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything. After you create combined lists for all topic areas, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product or system to be developed. The objective is to develop a consensus list of objects, services, constraints, and performance for the system to be built.

In many cases, an object or service described on a list will require further explanation. To accomplish this, stakeholders develop *mini-specifications* for

entries on the lists or by creating a use case (Section 8.4) that involves the object or service. For example, the mini-spec for the *SafeHome* object **Control Panel** might be:

> The control panel is a wall-mounted unit that is approximately 230 x 130 mm in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 75 x 75 mm OLED color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An *issues list* is maintained so that these ideas will be acted on later.

SAFEHOME

Conducting a Requirements-Gathering Meeting

The scene: A meeting room. The first requirements-gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

Many stakeholder concerns (e.g., accuracy, data accessibility, security) are the basis for nonfunctional system requirements (Section 8.2). As stakeholders enunciate these concerns, software engineers must consider them within the context

of the system to be built. Among the questions that must be answered [Lag10] are as follows:

- Can we build the system?
- Will this development process allow us to beat our competitors to market?
- Do adequate resources exist to build and maintain the proposed system?
- Will the system performance meet the needs of our customers?

The answers to these and other questions will evolve over time.

KEY POINT

QFD defines requirements in a way that maximizes customer satisfaction.

8.3.2 Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process" [Zul92]. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

ADVICE

Everyone wants to implement lots of exciting requirements, but be careful. That's how "requirements creep" sets in. On the other hand, exciting requirements lead to a breakthrough product!

Within the context of QFD, *normal requirements* identify the objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. *Expected requirements* are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. *Exciting requirements* go beyond the customer's expectations and prove to be very satisfying when present.

WebRef

Useful information on QFD can be obtained at **www.qfdi.org.**

Although QFD concepts can be applied across the entire software process [Par96a]; specific QFD techniques are applicable to the requirements elicitation activity. QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [Aka04].

8.3.3 Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* [Jac92], provide a description of how the system will be used. Use cases are discussed in greater detail in Section 8.4.

SAFEHOME

Developing a Preliminary User Scenario

The scene: A meeting room, continuing the first requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user ID and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

8.3.4 Elicitation Work Products

? What information is produced as a consequence of requirements gathering?

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include: (1) a statement of need and feasibility, (2) a bounded statement of scope for the system or product, (3) a list of customers, users, and other stakeholders who participated in requirements elicitation, (4) a description of the system's technical environment, (5) a list of requirements (preferably organized by function) and the domain constraints that applies to each, (6) a set of usage scenarios that provide insight into the use of the system or product under different operating conditions, and (7) any prototypes developed to better

define requirements. Each of these work products is reviewed by all people who have participated in requirements elicitation.

8.3.5 Agile Requirements Elicitation

KEY POINT

User stories are the way to document requirements elicited from customers in agile process models.

Within the context of an agile process, requirements are elicited by asking all stakeholders to create *user stories*. Each user story describes a simple system requirement written from the user's perspective. User stories can be written on small note cards, making it easy for developers to select and manage a subset of requirements to implement for the next product increment. Proponents claim that using note cards written in the user's own language allows developers to shift their focus to communication with stakeholders on the selected requirements rather than their own agenda [Mai10a].

Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that a consideration of overall business goals and nonfunctional requirements is often lacking. In some cases, rework is required to accommodate performance and security issues. In addition, user stories may not provide a sufficient basis for system evolution over time

8.3.6 Service-Oriented Methods

? What is a *service* in the context of service-oriented methods?

Service-oriented development views a system as an aggregation of services. A *service* can be "as simple as providing a single function, for example, a request/response-based mechanism that provides a series of random numbers, or can be an aggregation of complex elements, such as the Web service API" [Mic12].

Requirements elicitation in service-oriented development focuses on the definition of services to be rendered by an application. As a metaphor, consider the service provided when you visit a fine hotel. A doorperson greets guests. A valet parks their cars. The desk clerk checks the guests in. A bellhop manages the bags. The concierge assists guest with local arrangements. Each contact or *touchpoint* between a guest and a hotel employee is designed to enhance the hotel visit and represents a service offered.

KEY POINT

Requirements elicitation for service-oriented methods fines services render by an app. A touchpoint *represents an opportunity for the user to interact with the system to receive a desired service.*

Most service design methods emphasize understanding the customer, thinking creatively, and building solutions quickly [Mai10b]. To achieve these goals, requirements elicitation can include ethnographic studies,[11] innovation workshops, and early low-fidelity prototypes. Techniques for eliciting requirements must also acquire information about the brand and the stakeholders' perceptions of it. In addition to studying how the brand is used by customers, analysts need strategies to discover and document requirements about the desired qualities of new user experiences. User stories are helpful in this regard.

11 Studying user behavior in the environment where the proposed software product will be used.

The requirements for touchpoints should be characterized in a manner that indicates achievement of the overall service requirements. This suggests that each requirement should be traceable to a specific service.

8.4 Developing Use Cases

KEY POINT

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

In a book that discusses how to write effective use cases, Alistair Cockburn [Coc01b] notes that "a use case captures a contract . . . [that] describes the system's behavior under various conditions as the system responds to a request from one of its stakeholders . . ." In essence, a use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. Regardless of its form, a use case depicts the software or system from the end user's point of view.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

WebRef

An excellent paper on use cases can be downloaded from **www.ibm.com/developerworks/webservices/library/co-design7.html.**

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [Jac92] during the first iteration and secondary actors as more is learned about the system. *Primary actors* interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. *Secondary actors* support the system so that primary actors can do their work.

? What do I need to know in order to develop an effective use case?

Once actors have been identified, use cases can be developed. Jacobson [Jac92] suggests a number of questions[12] that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Recalling basic *SafeHome* requirements, we define four actors: **homeowner** (a user), **setup manager** (likely the same person as **homeowner,** but playing a different role), **sensors** (devices attached to the system), and the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function). For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC. The homeowner (1) enters a password to allow all other interactions, (2) inquires about the status of a security zone, (3) inquires about the status of a sensor, (4) presses the panic button in an emergency, and (5) activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:[13]

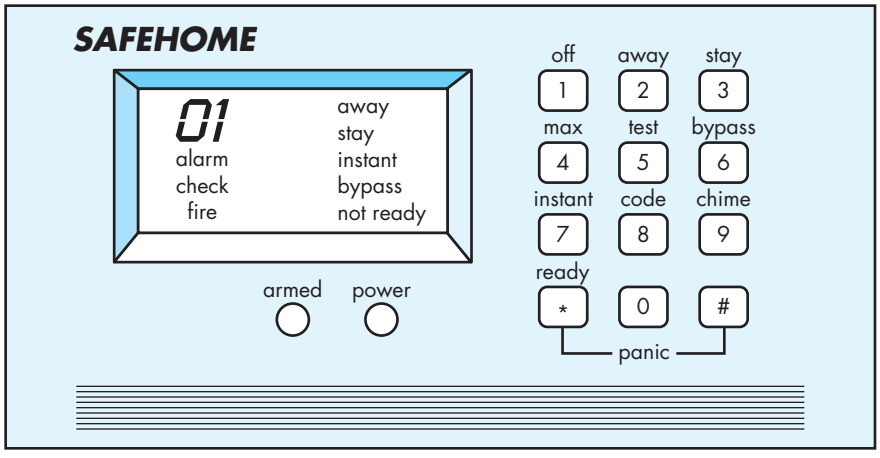
1. The homeowner observes the *SafeHome* control panel (Figure 8.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

12 Jacobson's questions have been extended to provide a more complete view of use case content.

13 Note that this use case differs from the situation in which the system is accessed via the Internet. In this case, interaction occurs via the control panel, not the GUI provided when a PC or mobile device is used.

FIGURE 8.1

***SafeHome* control panel**



2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 8.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.

ADVICE

Use cases are often written informally. However, use the template shown here to ensure that you've addressed all key issues.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case:	*InitiateMonitoring*
Primary actor:	Homeowner.
Goal in context:	To set the system to monitor sensors when the homeowner leaves the house or remains inside.
Preconditions:	System has been programmed for a password and to recognize various sensors.
Trigger:	The homeowner decides to "set" the system, that is, to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects "stay" or "away"
4. Homeowner: observes read alarm light to indicate that *SafeHome* has been armed

Exceptions:

1. Control panel is *not ready:* homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. *Stay* is selected: control panel beeps twice and a *stay* light is lit; perimeter sensors are activated.
5. *Away* is selected: control panel beeps three times and an *away* light is lit; all sensors are activated.

Priority:	Essential, must be implemented
When available:	First increment
Frequency of use:	Many times per day
Channel to actor:	Via control panel interface
Secondary actors:	Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

Use cases for other **homeowner** interactions would be developed in a similar manner. It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

SafeHome

Developing a High-Level Use Case Diagram

The scene: A meeting room, continuing the requirements-gathering meeting

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 8.2.)

Jamie: I'm just beginning to learn UML notation.[14] So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

Facilitator: Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

Doug: Is that legal in UML?

Facilitator: Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

Vinod: Okay, so we have use case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

Facilitator: Probably, but that can wait until we've considered other *SafeHome* functions.

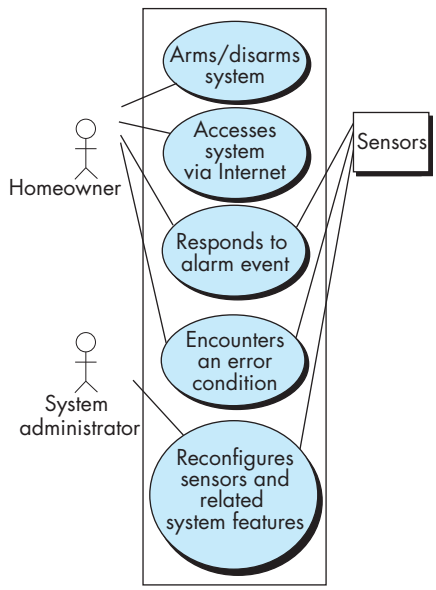
Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

Facilitator: Oh really. Tell me what we've missed.

(The meeting continues.)

Figure 8.2

UML use case diagram for *SafeHome* home security function



14 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

SOFTWARE TOOLS

Use Case Development

Objective: Assist in the development of use cases by providing automated templates and mechanisms for assessing clarity and consistency.

Mechanics: Tool mechanics vary. In general, use case tools provide fill-in-the-blank templates for creating effective use cases. Most use case functionality is embedded into a set of broader requirements engineering functions.

Representative Tools:[15]
The vast majority of UML-based analysis modeling tools provide both text and graphical support for use case development and modeling.
Objects by Design (**www.objectsbydesign.com/tools/umltools_byCompany.html**) provides comprehensive links to tools of this type.

8.5 BUILDING THE ANALYSIS MODEL[16]

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time. You should expect it to change.

As the analysis model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system. The analysis model and the methods that are used to build it are presented in detail in Chapters 9 to 11. We present a brief overview in the sections that follow.

ADVICE

It is always a good idea to get stakeholders involved. One of the best ways to do this is to have each stakeholder write use cases that describe how the software will be used.

8.5.1 Elements of the Analysis Model

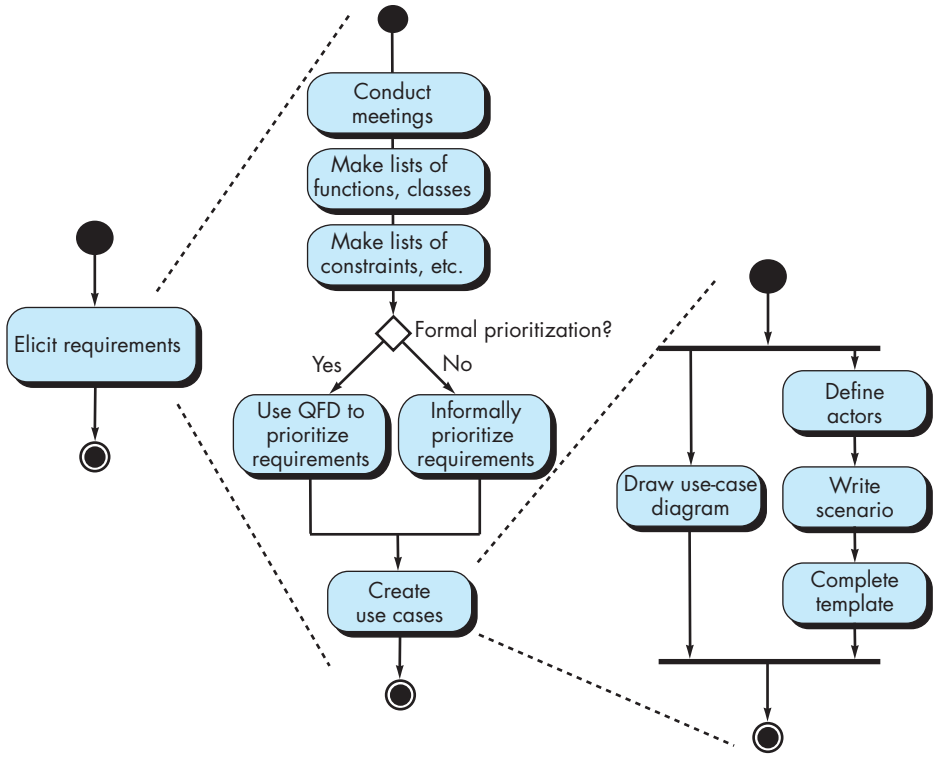
There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the analysis model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity. A set of generic elements is common to most analysis models.

15 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

16 Throughout this book, we use the terms *analysis model* and *requirements model* synonymously. Both refer to representations of the information, functional, and behavioral domains that describe problem requirements.

FIGURE 8.3

UML activity diagrams for eliciting requirements



ADVICE

One way to isolate classes is to look for descriptive nouns in a use case script. At least some of the nouns will be candidate classes. More on this in Chapter 12.

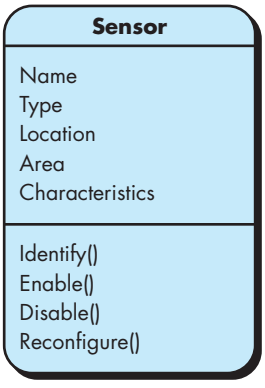
Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 8.4) and their corresponding use case diagrams (Figure 8.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 8.3 depicts a UML activity diagram[17] for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 8.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify, enable*) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with

17 A brief UML tutorial is presented in Appendix 1 for those who are unfamiliar with the notation.

FIGURE 8.4

Class diagram for sensor



one another and the relationships and interactions between classes. These are discussed in more detail in Chapter 10.

KEY POINT

A state is an externally observable mode of behavior. External stimuli cause transitions between states.

Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

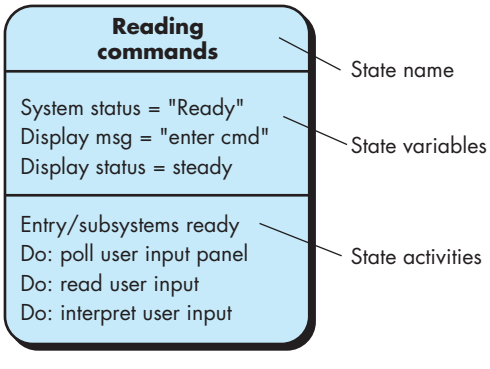
The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any observable mode of behavior. In addition, the state diagram indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

To illustrate the use of a state diagram, consider software embedded within the *SafeHome* control panel that is responsible for reading user input. A simplified UML state diagram is shown in Figure 8.5.

In addition to behavioral representations of the system as a whole, the behavior of individual classes can also be modeled. Further discussion of behavioral modeling is presented in Chapter 11.

FIGURE 8.5

UML state diagram notation



SafeHome

Preliminary Behavioral Modeling

The scene: A meeting room, continuing the requirements meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've just about finished talking about *SafeHome* home security functionality. But before we do, I want to discuss the behavior of the function.

Marketing person: I don't understand what you mean by behavior.

Ed (smiling): That's when you give the product a "timeout" if it misbehaves.

Facilitator: Not exactly. Let me explain.

(The facilitator explains the basics of behavioral modeling to the requirements gathering team.)

Marketing person: This seems a little technical. I'm not sure I can help here.

Facilitator: Sure you can. What behavior do you observe from the user's point of view?

Marketing person: Uh . . . well, the system will be *monitoring* the sensors. It'll be *reading commands* from the homeowner. It'll be *displaying* its status.

Facilitator: See, you can do it.

Jamie: It'll also be *polling* the PC to determine if there is any input from it, for example, Internet-based access or configuration information.

Vinod: Yeah, in fact, *configuring the system* is a state in its own right.

Doug: You guys are rolling. Let's give this a bit more thought . . . is there a way to diagram this stuff?

Facilitator: There is, but let's postpone that until after the meeting.

8.5.2 Analysis Patterns

ADVICE

If you want to obtain solutions to customer requirements more rapidly and provide your team with proven approaches, use analysis patterns.

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.[18] These *analysis patterns* [Fow97] suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler [Gey01] suggest two benefits that can be associated with the use of analysis patterns:

> First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations. Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements

18 In some cases, problems reoccur regardless of the application domain. For example, the features and functions used to solve user interface problems are common regardless of the application domain under consideration.

engineers can use search facilities to find and reuse them. Information about an analysis pattern (and other types of patterns) is presented in a standard template [Gey01][19] that is discussed in more detail in Chapter 16. Examples of analysis patterns and further discussion of this topic are presented in Chapter 11.

8.5.3 Agile Requirements Engineering

The intent of agile requirements engineering is to transfer ideas from stakeholders to the software team rather than create extensive analysis work products. In many situations, requirements are not predefined but emerge as each iteration of product development begins. As the agile team acquires a high-level understanding of a product's critical features use stories (Chapter 5) relevant to the next product increment are refined. The agile process encourages the early identification and implementation of the highest priority product features. This allows the early creation and testing of working prototypes.

Agile requirements engineering addresses important issues that are common in software projects: high requirements volatility, incomplete knowledge of development technology, and customers not able to articulate their visions until they see a working prototype. The agile process interleaves requirements engineering and design activities.

8.5.4 Requirements for Self-Adaptive Systems

? What are the characteristics of a self-adaptive system?

Self-adaptive systems[20] can reconfigure themselves, augment their functionality, protect themselves, recover from failure, and accomplish all of this while hiding most of their internal complexity from their users [Qur09]. Adaptive requirements document the variability needed for self-adaptive systems. This means that a requirement must encompass the notion of variability or flexibility while at the same time specifying either a functional or quality aspect of the software product. Variability might include timing uncertainty, user profile differences (e.g., end users versus systems administrators), behavior changes based on problem domain (e.g., commercial or educational), or predefined behaviors exploiting system assets.

Capturing adaptive requirements focuses on the same questions that are used for requirements engineering of more conventional systems. However, significant variability can be present when answering each of these questions. The more variable the answers, the more complex the resulting system will need to be to accommodate the requirements.

19 A variety of patterns templates have been proposed in the literature. If you have interest, see [Fow97], [Gam95], [Yac03], and [Bus07] among many sources.

20 An example of a self-adaptive system is a "location aware" app that adapts its behavior to the location of the mobile platform on which it resides.

8.6 Negotiating Requirements

Quote:

"A compromise is the art of dividing a cake in such a way that everyone believes he has the biggest piece."

Ludwig Erhard

WebRef

A brief paper on negotiation for software requirements can be downloaded from **www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html.**

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. Unfortunately, this rarely happens. In reality, you may have to enter into a *negotiation* with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market. The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a "win-win" result.[21] That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

INFO

The Art of Negotiation

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve, what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen to her. It's likely you'll gain knowledge that will help you to better negotiate your position.
4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle; commit to it and move on.

21 Dozens of books have been written on negotiating skills (e.g., [Fis11], [Lew09], [Rai06]). It is one of the more important skills that you can learn. Read one.

Fricker [Fri10] and his colleagues suggest replacing the traditional handoff of requirements specifications to software teams with a bidirectional communication process called *handshaking*. In handshaking, the software team proposes solutions to requirements, describes their impact, and communicates their intentions to customer representatives. The customer representatives review the proposed solutions, focusing on missing features and seeking clarification of novel requirements. Requirements are determined to be *good enough* if the customers accept the proposed solution.

Handshaking allows detailed requirements to be delegated to software teams. The teams need to elicit requirements from customers (e.g., product users and domain experts), thereby improving product acceptance. Handshaking tends to improve identification, analysis, and selection of variants and promotes win-win negotiation.

SafeHome

The Start of a Negotiation

The scene: Lisa Perez's office, after the first requirements gathering meeting.

The players: Doug Miller, software engineering manager and Lisa Perez, marketing manager.

The conversation:

Lisa: So, I hear the first meeting went really well.

Doug: Actually, it did. You sent some good people to the meeting . . . they really contributed.

Lisa (smiling): Yeah, they actually told me they got into it and it wasn't a "propeller head activity."

Doug (laughing): I'll be sure to take off my techie beanie the next time I visit . . . Look, Lisa, I think we may have a problem with getting all of the functionality for the home security system out by the dates your management is talking about. It's early, I know, but I've already been doing a little back-of-the-envelope planning and . . .

Lisa (frowning): We've got to have it by that date, Doug. What functionality are you talking about?

Doug: I figure we can get full home security functionality out by the drop-dead date, but we'll have to delay Internet access 'til the second release.

Lisa: Doug, it's the Internet access that gives *SafeHome* "gee whiz" appeal. We're going to build our entire marketing campaign around it. We've gotta have it!

Doug: I understand your situation, I really do. The problem is that in order to give you Internet access, we'll have to have a fully secure website up and running. That takes time and people. We'll also have to build a lot of additional functionality into the first release . . . I don't think we can do it with the resources we've got.

Lisa (still frowning): I see, but you've got to figure out a way to get it done. It's pivotal to home security functions and to other functions as well . . . those can wait until the next releases . . . I'll agree to that.

Lisa and Doug appear to be at an impasse, and yet they must negotiate a solution to this problem. Can they both "win" here? Playing the role of a mediator, what would you suggest?

8.7 Requirements Monitoring

Today, incremental development is commonplace. This means that use cases evolve, new test cases are developed for each new software increment, and continuous integration of source code occurs throughout a project. *Requirements*

monitoring can be extremely useful when incremental development is used. It encompasses five tasks: (1) *distributed debugging* uncovers errors and determines their cause, (2) *run-time verification* determines whether software matches its specification, (3) *run-time validation* assesses whether the evolving software meets user goals, (4) *business activity monitoring* evaluates whether a system satisfies business goals, and (5) *evolution and codesign* provides information to stakeholders as the system evolves.

Incremental development implies the need for incremental validation. Requirements monitoring supports continuous validation by analyzing user goal models against the system in use. For example, a monitoring system might continuously assess user satisfaction and use feedback to guide incremental improvements [Rob10].

8.8 Validating Requirements

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

? When I review requirements, what questions should I ask?

- Is each requirement consistent with the overall objectives for the system or product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

8.9 Avoiding Common Mistakes

Buschmann [Bus10] describes three related mistakes that must be avoided as a software team performs requirements engineering. He calls them: featuritis, flexibilitis, and performitis.

Fearturitis describes the practice of trading functional coverage for overall system quality. There is a tendency in some organizations to equate the quantity of functions delivered at the earliest possible time with the overall quality of the end product. This is driven in part by business stakeholders who think more is better. There is also a tendency of software developers to want to implement easy functions quickly without thought to their quality. The reality is that one of the most common causes of software project failure is lack of operational quality—*not* missing functionality. To avoid this trap, you should initiate a discussion (with other stakeholders) about the key functions the system requires and ensure that each delivered function exhibits all necessary quality attributes.

Flexibilitis happens when software engineers overload product with adaptation and configuration facilities. Overly flexible systems are hard to configure and exhibit poor operational performance. This can be a symptom of poorly defined system scope. The root cause, however, may be developers who use flexibility as a cover for uncertainty. Rather than making tough design decisions early, they provide design "hooks" to allow the addition of unplanned features. The result is a "flexible" system that is unnecessarily complex, more difficult to test, and more challenging to manage.

Performitis occurs when software developers become overly focused on system performance at the expense of quality attributes like maintainability, reliability, or security. System performance characteristics should be determined as part of an evaluation of nonfunctional software requirements. Performance should conform to the business need for a product and must be compatible with the other system characteristics.

8.10 Summary

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modeling activities that have been defined for the generic software

process. Seven distinct requirements engineering functions—inception, elicitation, elaboration, negotiation, specification, validation, and management—are conducted by members of the software team.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, QFD, and the development of usage scenarios.

Elaboration further expands requirements in a model—a collection of scenario-based, activity-based, class-based, behavioral, and flow-oriented elements. The model may reference analysis patterns, characteristics of the problem domain that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. In addition, each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built.

Problems and Points to Ponder

8.1. Why is it that many software developers don't pay enough attention to requirements engineering? Are there ever circumstances where you can skip it?

8.2. You have been given the responsibility to elicit requirements from a customer who tells you he is too busy to meet with you. What should you do?

8.3. Discuss some of the problems that occur when requirements must be elicited from three or four different customers.

8.4. Why do we say that the requirements model represents a snapshot of a system in time?

8.5. Let's assume that you've convinced the customer (you're a very good salesperson) to agree to every demand that you have as a developer. Does that make you a master negotiator? Why?

8.6. Develop at least three additional "context-free questions" that you might ask a stakeholder during inception.

8.7. Develop a requirements-gathering "kit." The kit should include a set of guidelines for conducting a requirements-gathering meeting and materials that can be used to facilitate the creation of lists and any other items that might help in defining requirements.

8.8. Your instructor will divide the class into groups of four or six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security function described in this chapter. Conduct a requirements-gathering meeting using the guidelines presented in this chapter.

8.9. Develop a complete use case for one of the following activities:

a. Making a withdrawal at an ATM.
b. Using your charge card for a meal at a restaurant.

c. Buying a stock using an online brokerage account.
d. Searching for books (on a specific topic) using an online bookstore.
e. An activity specified by your instructor.

8.10. What do use case "exceptions" represent?

8.11. Write a user story for one of the activities listed in question 8.9.

8.12. Consider the use case you created in question 8.9, write a nonfunctional requirement for the application.

8.13. Describe what *an analysis pattern* is in your own words.

8.14. Using the template presented in Section 8.5.2, suggest one or more analysis pattern for the following application domains:

a. Accounting software.
b. E-mail software.
c. Internet browsers.
d. Word-processing software.
e. Website creation software.
f. An application domain specified by your instructor.

8.15. What does *win-win* mean in the context of negotiation during the requirements engineering activity?

8.16. What do you think happens when requirement validation uncovers an error? Who is involved in correcting the error?

8.17. What five tasks make up a comprehensive requirements monitoring program?

Further Readings and Other Information Sources

Because it is pivotal to the successful creation of any complex computer-based system, requirements engineering is discussed in a wide array of books. Chemuturi (*Requirements Engineering and Management for Software Development Projects,* Springer, 2013) presents important aspects of requirements engineering. Pohl and Rupp (*Requirements Engineering Fundamentals,* Rocky Nook, 2011) present basic principles and concepts, and Pohl (*Requirements Engineering,* Springer, 2010) offers a more detailed view of the entire requirements engineering process. Young (*The Requirements Engineering Handbook,* Artech House Publishers, 2003) presents an in-depth discussion of requirements engineering tasks.

Beaty and Chen (*Visual Models for Software Products Best Practices,* Microsoft Press, 2012), Robertson (*Mastering the Requirements Process: Getting Requirements Right*, 3rd ed., Addison-Wesley, 2012), Hull and her colleagues (*Requirements Engineering,* 3rd ed., Springer-Verlag, 2010), Bray (*An Introduction to Requirements Engineering,* Addison-Wesley, 2002), Arlow (*Requirements Engineering,* Addison-Wesley, 2001), Gilb (*Requirements Engineering,* Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development,* Addison-Wesley, 1999), and Sommerville and Kotonya (*Requirement Engineering: Processes and Techniques,* Wiley, 1998), are but a few of many books dedicated to the subject. Wiegers (*More About Software Requirements,* Microsoft Press, 2010) provides many practical techniques for requirements gathering and management.

A patterns-based view of requirements engineering is described by Withall (*Software Requirement Patterns,* Microsoft Press, 2007). Ploesch (*Contracts, Scenarios and Prototypes,* Springer-Verlag, 2004) discusses advanced techniques for developing software requirements. Windle and Abreo (*Software Requirements Using the Unified Process,* Prentice Hall, 2002) discuss requirements engineering within the context of the Unified Process and UML notation. Alexander and Steven (*Writing Better Requirements,* Addison-Wesley, 2002) present a brief set of guidelines for writing clear requirements, representing them as scenarios, and reviewing the end result.

Use case modeling is often the driver for the creation of all other aspects of the analysis model. The subject is discussed at length by Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Leffingwell and his colleagues (*Managing Software Requirements: A Use Case Approach,* 2nd ed., Addison-Wesley, 2003) present a useful collection of requirement best practices.

A discussion of agile requirements can be found in books by Adzic (*Specification by Example: How Successful Teams Deliver the Right Software,* Manning Publications, 2011), Leffingwell (*Agile Requirements: Lean Requirements for Teams, Programs, and Enterprises,* Addison-Wesley, 2011), Cockburn (*Agile Software Development: The Cooperative Game,* 2nd ed., Addison-Wesley, 2006), and Cohn (*User Stories Applied: For Agile Software Development,* Addison-Wesley, 2004).

A wide variety of information sources on requirements engineering and analysis is available on the Internet. An up-to-date list of World Wide Web references that are relevant to requirements engineering and analysis can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

9 Requirements Modeling: Scenario-Based Methods

Key Concepts

activity diagram . . 180
domain analysis . . 170
formal use case. . . 177
requirements analysis 167
requirements modeling 171
scenario-based modeling 173
swimlane diagram. 181

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model[1]—actually a set of models—is the first technical representation of a system.

In a seminal book on requirements modeling methods, Tom DeMarco [DeM79] describes the process in this way:

> Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals. The products of analysis must be highly maintainable. This applies

Quick Look

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an analyst) builds the model using requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. In this chapter you'll consider requirements modeling from a scenario-based perspective and examine how UML can be used to supplement the scenarios. In Chapters 10 and 11, you'll learn about other "dimensions" of the requirements model. By examining a number of different dimensions, you'll increase the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Scenario-based modeling represents the system from the user's point of view. By building a scenario-based model, you will be able to better understand how the user interacts with the software, uncovering the major functions and features that stakeholder require of the system.

What is the work product? Scenario-based modeling produces a text-oriented representation call a "use case." The use case describes a specific interaction in a manner that can be informal (a simple narrative) or more structured and formal in nature. The use case can be supplemented with a number of different UML diagrams that overlay a more procedural view of the interaction.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

1 In earlier editions of this book, the term *analysis model* was used, rather than *requirements model*. In this edition, we've decided to use both phrases to represent the modeling activity that defines various aspects of the problem to be solved. *Analysis* is the action that occurs as *requirements* are derived.

uml models 179
use cases 173
use case exception 177

> particularly to the Target Document [software requirements specification]. Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out. Graphics have to be used whenever possible. We have to differentiate between logical [essential] and physical [implementation] considerations . . . At the very least, we need . . . Something to help us partition our requirements and document that partitioning before specification . . . Some means of keeping track of and evaluating interfaces . . . New tools to describe logic and policy, something better than narrative text

Although DeMarco wrote about the attributes of analysis modeling more than three decades ago, his comments still apply to modern requirements modeling methods and notation.

9.1 Requirements Analysis

Quote:

"Any one 'view' of requirements is insufficient to understand or describe the desired behavior of a complex system."

Alan M. Davis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer,* an *analyst,* or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering (Chapter 8).

The requirements modeling action results in one or more of the following types of models:

- *Scenario-based models* of requirements from the point of view of various system "actors."
- *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements.
- *Behavioral and patterns-based models* that depict how the software behaves as a consequence of external "events."
- *Data models* that depict the information domain for the problem.
- *Flow-oriented models* that represent the functional elements of the system and how they transform data as they move through the system.

KEY POINT

The analysis model and requirements specification provide a means for assessing quality once the software is built.

These models provide a software designer with information that can be translated to architectural-, interface-, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

In this chapter, we focus on *scenario-based modeling*—a technique that is growing increasingly popular throughout the software engineering community. In Chapters 10 and 11 we consider class-based models and behavioral models. Over the past decade, flow and data modeling have become less commonly used, while scenario and class-based methods, supplemented with behavioral approaches and pattern-based techniques have grown in popularity.[2]

Quote:

"Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are need."

Andrew Hunt and David Thomas

9.1.1 Overall Objectives and Philosophy

Throughout analysis modeling, your primary focus is on *what,* not *how.* What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?[3]

In previous chapters, we noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.[4]

KEY POINT

The analysis model should describe what the customer wants, establish a basis for design, and establish a target for validation.

The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design (Chapters 12 through 18) that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 9.1.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

---

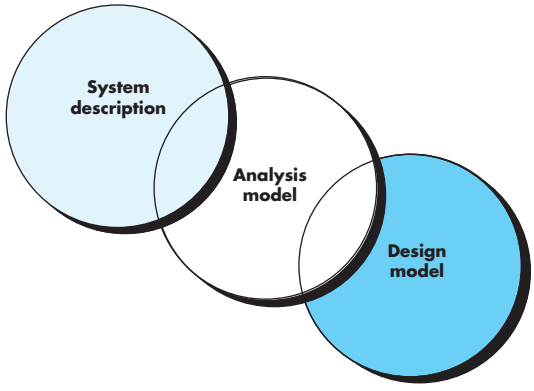
2 Our presentation of flow-oriented modeling and data modeling has been omitted from this edition. However, copious information about these older requirements modeling methods can be found on the Web. If you have interest, use the search phrase "structured analysis."

3 It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of *how* as well as *what.* However, the primary focus should remain on *what.*

4 Alternatively, the software team may choose to create a prototype (Chapter 4) in an effort to better understand requirements for the system.

FIGURE 9.1

The requirements model as a bridge between the system description and the design model



9.1.2 Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

? Are there some basic guidelines that can guide us as we do requirements analysis work?

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" [Arl02] that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, efforts should be made to reduce it.
- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.
- *Keep the model as simple as it can be.* Don't add additional diagrams when they add no new information. Don't use complex notational forms when a simple list will do.

Quote:

"Problems worthy of attack, prove their worth by hitting back."

Piet Hein

9.1.3 Domain Analysis

WebRef

Many useful resources for domain analysis and many other topics can be found at **http://www.sei.cmu.edu/**.

In the discussion of requirements engineering (Chapter 8), we noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis*. Firesmith [Fir93] describes domain analysis in the following way:

> Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

KEY POINT

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides. The intent is to identify common problem solving elements that are applicable to all applications within the domain.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.[5]

Using terminology that was introduced previously in this book, domain analysis may be viewed as an umbrella activity for the software process. By this we mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst[6] is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

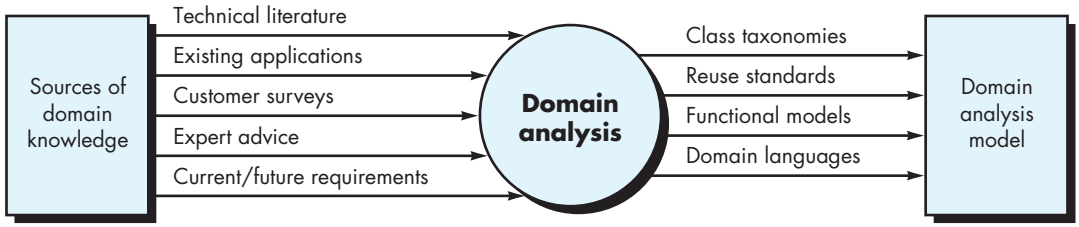
Figure 9.2 [Arn89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

---

5 A complementary view of domain analysis "involves modeling the domain so that software engineers and other stakeholders can better learn about it . . . not all domain classes necessarily result in the development of reusable classes." [Let03a]

6 Do not make the assumption that because a domain analyst is at work, a software engineer need not understand the application domain. Every member of a software team should have some understanding of the domain in which the software is to be placed.

FIGURE 9.2 Input and output for domain analysis



SAFEHOME

Domain Analysis

The scene: Doug Miller's office, after a meeting with marketing.

The players: Doug Miller, software engineering manager, and Vinod Raman, a member of the software engineering team.

The conversation:

Doug: I need you for a special project, Vinod. I'm going to pull you out of the requirements-gathering meetings.

Vinod (frowning): Too bad. That format actually works . . . I was getting something out of it. What's up?

Doug: Jamie and Ed will cover for you. Anyway, marketing insists that we deliver the Internet capability along with the home security function in the first release of *SafeHome*. We're under the gun on this . . . not enough time or people, so we've got to solve both problems—the PC interface and the Web interface—at once.

Vinod (looking confused): I didn't know the plan was set . . . we're not even finished with requirements gathering.

Doug (a wan smile): I know, but the time lines are so short that I decided to begin strategizing with marketing right now . . . anyhow, we'll revisit any tentative plan once we have the info from all of the requirements-gathering meetings.

Vinod: Okay, what's up? What do you want me to do?

Doug: Do you know what "domain analysis" is?

Vinod: Sort of. You look for similar patterns in Apps that do the same kinds of things as the App you're building. If possible, you then steal the patterns and reuse them in your work.

Doug: Not sure I like the word *steal,* but basically you have it right. What I'd like you to do is to begin researching existing user interfaces for systems that control something like *SafeHome*. I want you to propose a set of patterns and analysis classes that can be common to both the PC-based interface that'll sit in the house and the browser-based interface that is accessible via the Internet.

Vinod: We can save time by making them the same . . . why don't we just do that?

Doug: Ah . . . it's nice to have people who think like you do. That's the whole point—we can save time and effort if both interfaces are nearly identical, implemented with the same code, blah, blah, that marketing insists on.

Vinod: So you want, what—classes, analysis patterns, design patterns?

Doug: All of 'em. Nothing formal at this point. I just want to get a head start on our internal analysis and design work.

Vinod: I'll go to our class library and see what we've got. I'll also use a patterns template I saw in a book I was reading a few months back.

Doug: Good. Go to work.

9.1.4 Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that

Quote:

"[A]nalysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you."

Tom DeMarco

manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process (Chapter 4) are predominantly object oriented.

In this edition of the book, we have chosen to emphasize elements of object-oriented analysis as it is modeled using UML. Our goal is to suggest a combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design.

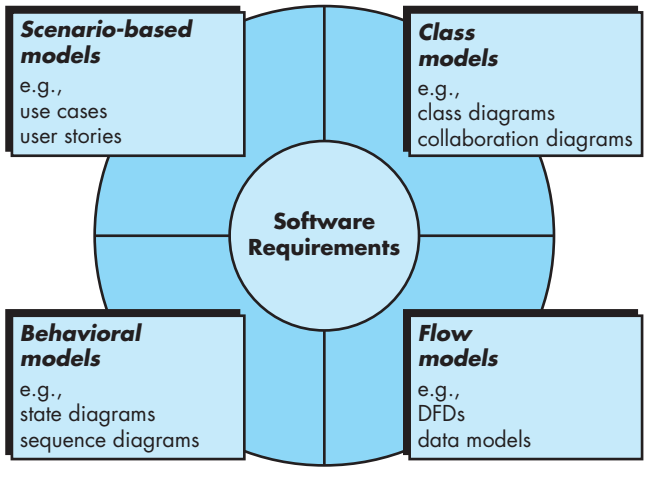
Each element of the requirements model (Figure 9.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

? What different points of view can be used to describe the requirements model?

Analysis modeling leads to the derivation of one or more of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

FIGURE 9.3

Elements of the analysis model



9.2 Scenario-Based Modeling

Although the success of a computer-based system or product is measured in many ways, user satisfaction resides at the top of the list. If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML[7] begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

9.2.1 Creating a Preliminary Use Case

Quote:

"[Use cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use cases)."

Ivar Jacobson

Alistair Cockburn characterizes a use case as a "contract for behavior" [Coc01b]. As we discussed in Chapter 8, the "contract" defines the way in which an actor[8] uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. In this section, we examine how use cases are developed as part of the analysis modeling activity.[9]

In Chapter 8, we noted that a use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description? These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

ADVICE

In some situations, use cases become the dominant requirements engineering mechanism. However, this does not mean that you should discard other modeling methods when they are appropriate.

What to Write About? The first two requirements engineering tasks—inception and elicitation—provide you with the information you'll need to begin writing use cases. Requirements-gathering meetings, quality function deployment (QFD), and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams (Section 9.3.1) developed as part of requirements modeling.

---

7 UML will be used as the modeling notation throughout this book. Appendix 1 provides a brief tutorial for those readers who may be unfamiliar with basic UML notation.

8 An actor is not a specific person, but rather a role that a person (or a device) plays within a specific context. An actor "calls on the system to deliver one of its services " [Coc01b].

9 Use cases are a particularly important part of analysis modeling for user interfaces. Interface analysis and design is discussed in detail in Chapter 15.

SafeHome

Developing Another Preliminary User Scenario

The scene: A meeting room, during the second requirements-gathering meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: It's time that we begin talking about the *SafeHome* surveillance function. Let's develop a user scenario for access to the surveillance function.

Jamie: Who plays the role of the actor on this?

Facilitator: I think Meredith (a marketing person) has been working on that functionality. Why don't you play the role?

Meredith: You want to do it the same way we did it last time, right?

Facilitator: Right . . . same way.

Meredith: Well, obviously the reason for surveillance is to allow the homeowner to check out the house while he or she is away, to record and play back video that is captured . . . that sort of thing.

Ed: Will we use compression to store the video?

Facilitator: Good question, Ed, but let's postpone implementation issues for now. Meredith?

Meredith: Okay, so basically there are two parts to the surveillance function . . . the first configures the system including laying out a floor plan—we have to have tools to help the homeowner do this—and the second part is the actual surveillance function itself. Since the layout is part of the configuration activity, I'll focus on the surveillance function.

Facilitator (smiling): Took the words right out of my mouth.

Meredith: Um . . . I want to gain access to the surveillance function either via the PC or via the Internet. My feeling is that the Internet access would be more frequently used. Anyway, I want to be able to display camera views on a PC and control pan and zoom for a specific camera. I specify the camera by selecting it from the house floor plan. I want to selectively record camera output and replay camera output. I also want to be able to block access to one or more cameras with a specific password. I also want the option of seeing small windows that show views from all cameras and then be able to pick the one I want enlarged.

Jamie: Those are called thumbnail views.

Meredith: Okay, then I want thumbnail views of all the cameras. I also want the interface for the surveillance function to have the same look and feel as all other *SafeHome* interfaces. I want it to be intuitive, meaning I don't want to have to read a manual to use it.

Facilitator: Good job. Now, let's go into this function in a bit more detail . . .

The *SafeHome* home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the **homeowner** actor:

- Select camera to view.
- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements-gathering team develops use cases for each of the functions noted. In general, use cases are written first in an informal narrative fashion. If more formality is required, the same use case is rewritten using a structured format similar to the one proposed in Chapter 8 and reproduced later in this section as a sidebar.

To illustrate, consider the function *access camera surveillance via the Internet—display camera views* (**ACS-DCV**). The stakeholder who takes on the role of the **homeowner** actor might write the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence. Revisiting the **ACS-DCV** function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.

Quote:

"Use cases can be used in many [software] processes. Our favorite is a process that is iterative and risk driven."

Geri Schneider and Jason Winters

10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

It is important to note that this sequential presentation does not consider any alternative interactions (the narrative is more free flowing and did represent a few alternatives). Use cases of this type are sometimes referred to as *primary scenarios* [Sch98a].

9.2.2 Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions [Sch98a]:

? How do I examine alternative courses of action when I develop a use case?

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Answers to these questions result in the creation of a set of *secondary scenarios* that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.

Can the actor take some other action at this point? The answer is yes. Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

Is it possible that the actor will encounter some error condition at this point? Any number of error conditions can occur as a computer-based system operates. In this context, we consider only error conditions that are likely as a direct result of the action described in step 6 or step 7. Again the answer to the question is yes. A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."[10] This error condition becomes a secondary scenario.

10 In this case, another actor, the **system administrator,** would have to configure the floor plan, install and initialize (e.g., assign an equipment ID) all cameras, and test each camera to be certain that it is accessible via the system and through the floor plan.

Is it possible that the actor will encounter some other behavior at this point? Again the answer to the question is yes. As steps 6 and 7 occur, the system may encounter an alarm condition. This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the **ACS-DCV** use case. Rather, a separate use case—**Alarm condition encountered**—would be developed and referenced from other use cases as required.

? What is a use case exception and how do I determine what exceptions are likely?

Each of the situations described in the preceding paragraphs is characterized as a use case exception. An *exception* describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.

Cockburn [Coc01b] recommends a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

The list of extensions developed as a consequence of asking and answering these questions should be "rationalized" [Co01b] using the following criteria: an exception should be noted within the use case if the software can detect the condition described and then handle the condition once it has been detected. In some cases, an exception will precipitate the development of another use case (to handle the condition noted).

9.2.3 Writing a Formal Use Case

The informal use cases presented in Section 9.2.1 are sometimes sufficient for requirements modeling. However, when a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The **ACS-DCV** use case shown in the sidebar follows a typical outline for formal use cases. The *goal in context* identifies the overall scope of the use case.

The *precondition* describes what is known to be true before the use case is initiated. The *trigger* identifies the event or condition that "gets the use case started" [Coc01b]. The *scenario* lists the specific actions that are required by the actor and the appropriate system responses. *Exceptions* identify the situations uncovered as the preliminary use case is refined (Section 9.2.2). Additional headings may or may not be included and are reasonably self-explanatory.

SafeHome

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Infrequent.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

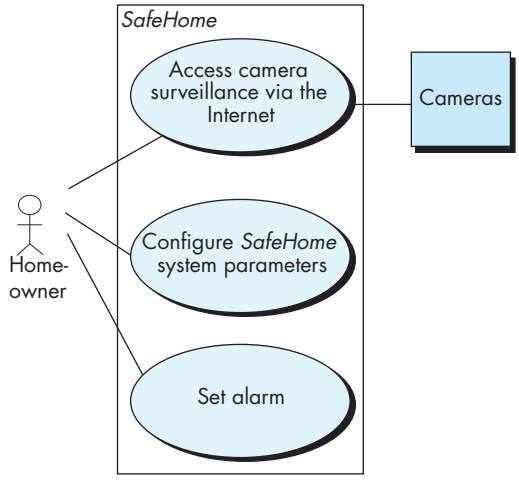
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

FIGURE 9.4

Preliminary use case diagram for the *SafeHome* system



WebRef

When are you finished writing use cases? For a worthwhile discussion of this topic, see **ootips.org/use-cases-done.html**.

In many cases, there is no need to create a graphical representation of a usage scenario. However, diagrammatic representation can facilitate understanding, particularly when the scenario is complex. As we noted earlier in this book, UML does provide use case diagramming capability. Figure 9.4 depicts a preliminary use case diagram for the *SafeHome* product. Each use case is represented by an oval. Only the **ACS-DCV** use case has been discussed in this section.

Every modeling notation has limitations, and the use case is no exception. Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous. A use case focuses on function and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer. If developed properly, the use case can provide substantial benefit as a modeling tool.

9.3 UML Models That Supplement the Use Case

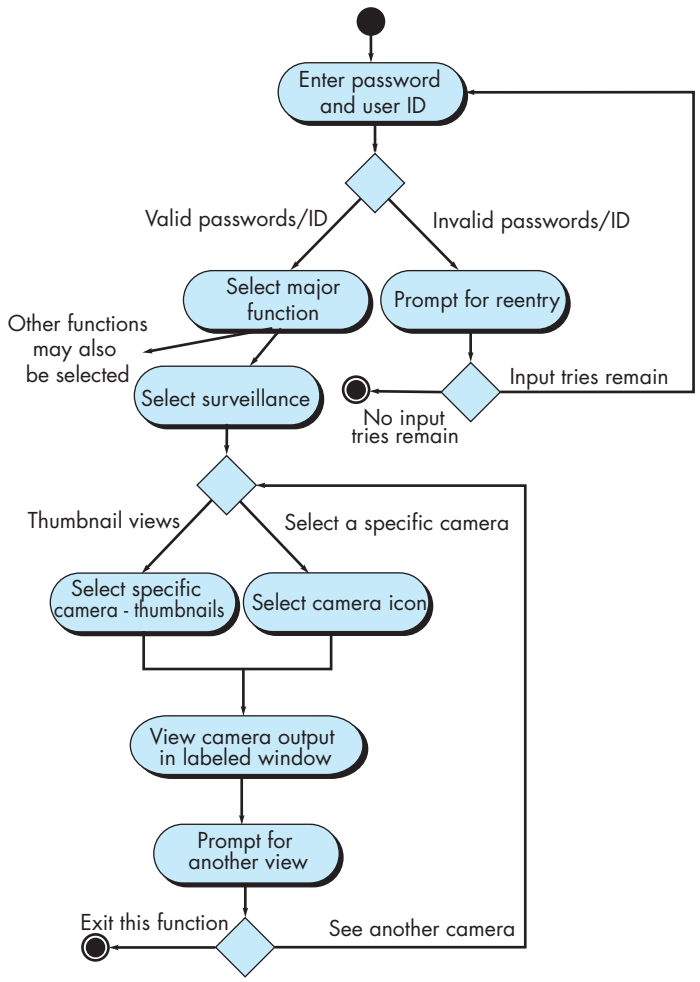
There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner. In such cases, you can choose from a broad array of UML graphical models.

9.3.1 Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the **ACS-DCV** use case is shown in Figure 9.5. It should be noted that the activity diagram adds additional detail not directly mentioned (but implied) by the use case. For example, a user may only attempt to enter **userID** and **password** a limited number of times. This is represented by a decision diamond below "Prompt for reentry."

FIGURE 9.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.



9.3.2 Swimlane Diagrams

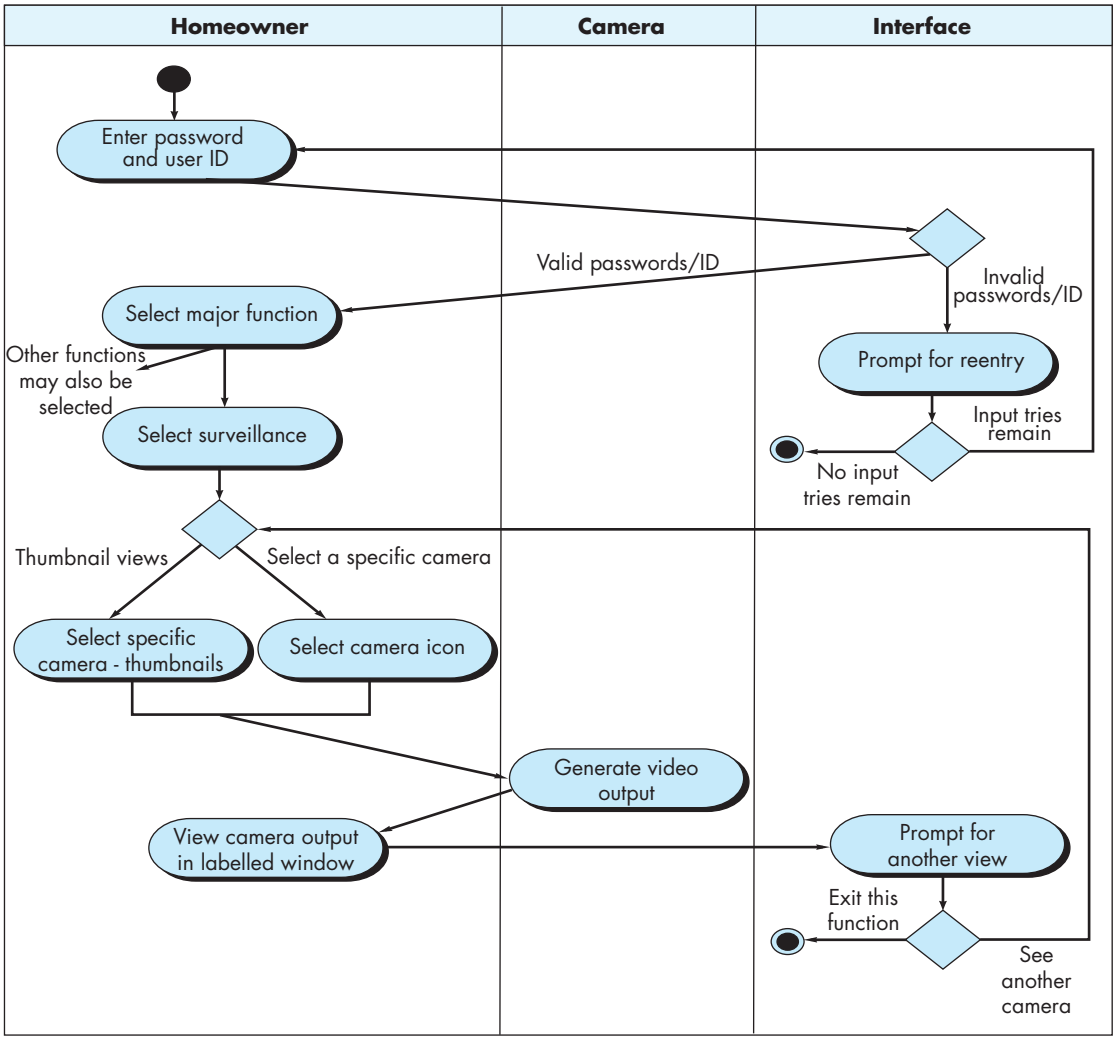
KEY POINT

A UML swimlane diagram represents the flow of actions and decisions and indicates which actors perform each.

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor (if there are multiple actors involved in a specific use case) or analysis class (Chapter 10) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—**Homeowner**, **Camera**, and **Interface**—have direct or indirect responsibilities in the context of the activity diagram represented in Figure 9.5. Referring to Figure 9.6, the activity diagram is rearranged so that

FIGURE 9.6 **Swimlane diagram for Access camera surveillance via the Internet—display camera views function.**



activities associated with a particular analysis class fall inside the swimlane for that class. For example, the **Interface** class represents the user interface as seen by the homeowner. The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the **Interface** swimlane. However, arrows lead from that swimlane back to the **Homeowner** swimlane, where homeowner actions occur.

Quote:

"A good model guides your thinking, a bad one warps it."

Brian Marick

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions (or other procedural steps) to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system In Chapters 10 and 11, we examine other dimensions of requirements modeling.

9.4 Summary

The objective of requirements modeling is to create a variety of representations that describe what the customer requires, establish a basis for the creation of a software design, and define a set of requirements that can be validated once the software is built. The requirements model bridges the gap between a system-level description that describes overall system and business functionality and a software design that describes the software's application architecture, user interface, and component-level structure.

Scenario-based models depict software requirements from the user's point of view. The use case—a narrative or template-driven description of an interaction between an actor and the software—is the primary modeling element. Derived during requirements elicitation, the use case defines the keys steps for a specific function or interaction. The degree of use case formality and detail varies, but the end result provides necessary input to all other analysis modeling activities. Scenarios can also be described using an activity diagram—a flowchart-like graphical representation that depicts the processing flow within a specific scenario. Swimlane diagrams illustrate how the processing flow is allocated to various actors or classes.

Problems and Points to Ponder

9.1. Is it possible to begin coding immediately after a requirements model has been created? Explain your answer and then argue the counterpoint.

9.2. An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are *not* visible in these domains? Provide a few examples.

9.3. What is the purpose of domain analysis? How is it related to the concept of requirements patterns?

9.4. Is it possible to develop an effective analysis model without developing all four elements shown in Figure 9.3? Explain.

9.5. The department of public works for a large city has decided to develop a Web-based pothole tracking and repair system (PHTRS). A description follows:

> Citizens can log onto a website and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and include pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used, and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, and dollar amount of damage. PHTRS is an online system; all queries are to be made interactively.

Draw a UML use case diagram PHTRS system. You'll have to make a number of assumptions about the manner in which a user interacts with this system.

9.6. Write two or three use cases that describe the roles of various actors in the PHTRS described in Problem 9.5.

9.7. Develop an activity diagram for one aspect of PHTRS.

9.8. Develop a swimlane diagram for one or more aspects of PHTRS.

Further Readings and Information Sources

Use cases can serve as the foundation for all requirements modeling approaches. The subject is discussed at length by Gomaa (*Software Modeling: UML, Use Case, Patterns, and Architecture,* Cambridge University Press, 2011), Rosenberg and Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice,* Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality,* Addison-Wesley, 2005), Alexander and Maiden (Eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle,* Wiley, 2004), Bittner and Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01b], and other references noted in Chapter 8.

UML modeling techniques that can be applied for both analysis and design are discussed by Dennis and his colleagues (*Systems Analysis and Design with UML Version 2.0,* 4th ed., Wiley, 2012), O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0,* Wiley, 2005), Arlow and Neustadt (*UML 2 and the Unified Process,* 2nd ed., Addison-Wesley, 2005), Roques (*UML in Practice,* Wiley, 2004), Larman (*Applying UML and Patterns,* 2nd ed., Prentice Hall, 2001), and Rosenberg and Scott (*Use Case Driven Object Modeling with UML,* Addison-Wesley, 1999).

Some books on requirements include Robertson and Robertson (*Mastering the Requirements Process: Getting Requirements Right,* 3rd ed., Addison-Wesley, 2012), Hull, Jackson, and Dick (*Requirements Engineering,* 3rd ed., Springer, 2010), and Alexander and Beus-Dukic (*Discovering Requirements: How to Specify Products and Services,* Wiley, 2009). A wide variety of information sources on requirements modeling are available on the Internet.

An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

10 Requirements Modeling: Class-Based Methods

Key Concepts

analysis classes . . 185
analysis packages . 199
associations 198
attributes 188
collaborations 195
CRC modeling 192
dependencies 198
grammatical parse. 185
operations 189
responsibilities . . . 193

When they were first introduced in the early 1990s, class-based methods for requirements modeling were often categorized as *object-oriented analysis*. Although a number of different class-based methods and representations were introduced, Coad and Yourdon [Coa91] noted one universal characteristic for all of them:

> [Object-oriented methods are all] based upon concepts that we first learned in kindergarten: objects and attributes, wholes and parts, classes and members.

Class-based methods for requirements modeling use these common concepts to craft a representation of an application that can be understood by nontechnical stakeholders. As the requirements model is refined and expanding, it evolves into a specification that can be used by software engineers in the creation of the software design.

Class-based modeling represents the objects that the system will manipulate, the operations (also called *methods* or *services*) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are

Quick Look

What is it? Software problems can almost always be characterized in terms of a set of interacting objects each representing something of interest within a system. Each object becomes a member of a class of objects. Each object is described by its state—the data attributes that describe the object. All of this can be represented using class-based requirements modeling methods.

Who does it? A software engineer (sometimes called an analyst) builds the class-based model using requirements elicited from the customer.

Why is it important? A class-based requirements model makes use of objects drawn from the customer's view of an application or system. The model depicts a view of the system that is common to the customer. Therefore, it can be readily evaluated by the customer, resulting in useful feedback at the earliest possible time. Later, as the model is refined, it becomes the basis for software design.

What are the steps? Class-based modeling defines objects, attributes, and relationships. A set of simple heuristics can be developed to extract objects and classes from a problem statement and then represent them in text-based and/or diagrammatic forms. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency.

What is the work product? A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

defined. The elements of a class-based model include classes and objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The sections that follow present a series of informal guidelines that will assist in their identification and representation.

10.1 Identifying Analysis Classes

Quote:

"The really hard problem is discovering what are the right objects [classes] in the first place."

Carl Argila

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you "look around" the problem space of a software application, the classes (and objects) may be more difficult to comprehend.

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model (Chapter 9) and performing a "grammatical parse" [Abb83] on the use cases developed for the system to be built. Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

But what should we look for once all of the nouns have been isolated? *Analysis classes* manifest themselves in one of the following ways:

? How do analysis classes manifest themselves as elements of the solution space?

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

This categorization is but one of many that have been proposed in the literature.[1] For example, Budd [Bud96] suggests a taxonomy of classes that includes

1 Another important categorization, defining entity, boundary, and controller classes, is discussed in Section 10.5.

producers (sources) and *consumers* (sinks) of data, *data managers*, *view* or *observer classes*, and *helper classes*.

It is also important to note what classes or objects are not. In general, a class should never have an "imperative procedural name" [Cas89]. For example, if the developers of software for a medical imaging system defined an object with the name **InvertImage** or even **ImageInversion,** they would be making a subtle mistake. The **Image** obtained from the software could, of course, be a class (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the object. It is likely that inversion would be defined as an operation for the object **Image,** but it would not be defined as a separate class to connote "image inversion." As Cashman [Cas89] states, "[T]he intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data."

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse (nouns are underlined, verbs italicized) for a processing narrative[2] for the *SafeHome* security function.

> The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC or a control panel.
>
> During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.
>
> When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained*.
>
> The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

ADVICE

The grammatical parse is not foolproof, but it can provide you with an excellent jump start if you're struggling to define data objects and the transforms that operate on them.

2 A processing narrative is similar to the use case in style but somewhat different in purpose. The processing narrative provides an overall description of the function to be developed. It is not a scenario written from one actor's point of view. It is important to note, however, that a grammatical parse can also be used for every use case developed as part of requirements gathering (elicitation).

Extracting the nouns, we can propose a number of potential classes:

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that we call each entry in the list a "potential" object. We must consider each further before a final decision is made.

? How do I determine whether a potential class should, in fact, become an analysis class?

Coad and Yourdon [Coa91] suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

1. *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. *Multiple attributes.* During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. *Common operations.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

Quote:

"Classes struggle, some classes triumph, others are eliminated."

Mao Zedong

To be considered a legitimate class for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. The decision for inclusion of potential classes in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated.

However, the first step of class-based modeling is the definition of classes, and decisions (even subjective ones) must be made. With this in mind, you should apply the selection characteristics to the list of potential *SafeHome* classes:

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that (1) the preceding list is not all inclusive, additional classes would have to be added to complete the model; (2) some of the rejected potential classes will become attributes for those classes that were accepted (e.g., number and type are attributes of **Sensor,** and master password and telephone number may become attributes of **System**); (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **Homeowner** class would satisfy characteristics 1 and 2 and would have been accepted).

10.2 Specifying Attributes

KEY POINT

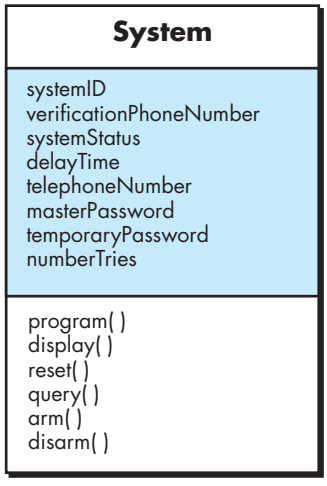
Attributes are the set of data objects that fully define the class within the context of the problem.

Attributes describe a class that has been selected for inclusion in the analysis model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be quite different than the attributes of the same class when it is used in the context of the professional baseball pension system. In the former, attributes such as name, position, batting average, fielding percentage, years played, and games played might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class. In addition, the following question should be answered for each class: *What data items (composite and/or elementary) fully define this class in the context of the problem at hand?*

FIGURE 10.1

Class diagram for the system class



To illustrate, we consider the **System** class defined for *SafeHome*. A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. We can represent these composite data items in the following manner:

identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the **System** class (shaded portion of Figure 10.1).

Sensors are part of the overall *SafeHome* system, and yet they are not listed as data items or as attributes in Figure 10.1. **Sensor** has already been defined as a class, and multiple **Sensor** objects will be associated with the **System** class. In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

10.3 DEFINING OPERATIONS

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by

ADVICE

When you define operations for an analysis class, focus on problem-oriented behavior rather than behaviors required for implementation.

operating on attributes and/or associations (Section 10.5). Therefore, an operation must have "knowledge" of the nature of the class attributes and associations.

As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific class. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is *assigned* a number and type" or "a master password is *programmed* for *arming and disarming* the system." These phrases indicate a number of things:

- That an *assign()* operation is relevant for the **Sensor** class.
- That a *program()* operation will be applied to the **System** class.
- That *arm()* and *disarm()* are operations that apply to **System** class.

Upon further investigation, it is likely that the operation *program()* will be divided into a number of more specific suboperations required to configure the system. For example, *program()* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program()* as a single operation.

In addition to the grammatical parse, you can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, we explore this matter in a bit more detail.

SafeHome

Class Models

The scene: Ed's cubicle, as analysis modeling begins.

The players: Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

The conversation:
[Ed has been working to extract classes from the use case template for ACS-DCV (presented in an earlier sidebar in this chapter) and is presenting the classes he has extracted to his colleagues.]

Ed: So when the homeowner wants to pick a camera, he or she has to pick it from a floor plan. I've defined a **FloorPlan** class. Here's the diagram.
(They look at Figure 10.2.)

Jamie: So **FloorPlan** is an object that is put together with walls, doors, windows, and cameras. That's what those labeled lines mean, right?

Ed: Yeah, they're called "associations." One class is associated with another according to the associations I've shown. [Associations are discussed in Section 10.5.]

Vinod: So the actual floor plan is made up of walls and contains cameras and sensors that are placed within those walls. How does the floor plan know where to put those objects?

Ed: It doesn't, but the other classes do. See the attributes under, say, **WallSegment,** which is used to build a wall. The wall segment has start and stop coordinates and the *draw()* operation does the rest.

Jamie: And the same goes for windows and doors. Looks like camera has a few extra attributes.

Ed: Yeah, I need them to provide pan and zoom info.

Vinod: I have a question. Why does the camera have an ID but the others don't? I notice you have an attribute

called **nextWall**. How will **WallSegment** know what the next wall will be?

Ed: Good question, but as they say, that's a design decision, so I'm going to delay that until . . .

Jamie: Give me a break . . . I'll bet you've already figured it out.

Ed (smiling sheepishly): True, I'm gonna use a list structure which I'll model when we get to design. If you get religious about separating analysis and design, the level of detail I have right here could be suspect.

Jamie: Looks pretty good to me, but I have a few more questions.

(Jamie asks questions which result in minor modifications.)

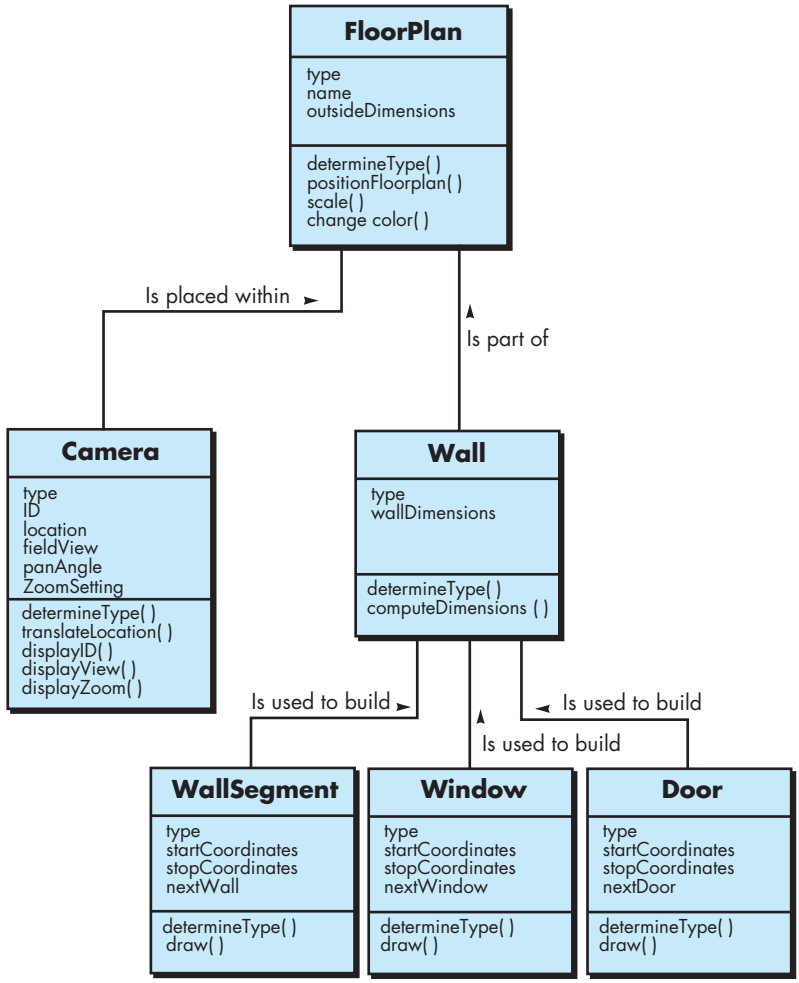
Vinod: Do you have CRC cards for each of the objects? If so, we ought to role-play through them, just to make sure nothing has been omitted.

Ed: I'm not quite sure how to do them.

Vinod: It's not hard and they really pay off. I'll show you.

FIGURE 10.2

Class diagram for FloorPlan (see sidebar discussion)



10.4 Class-Responsibility-Collaborator Modeling

Quote:

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code."

C. Horstmann

Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:

> A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [Amb95]. *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action.

WebRef

An excellent discussion of these class types can be found at **http://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf.**

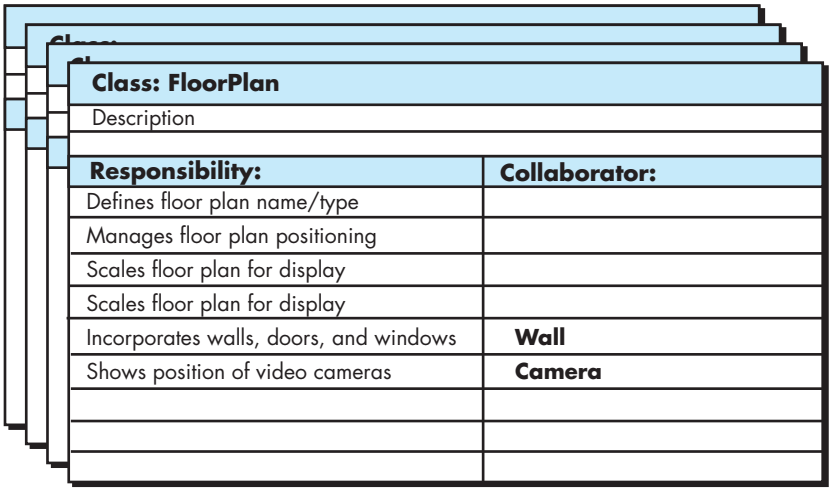
A simple CRC index card for the **FloorPlan** class is illustrated in Figure 10.3. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 10.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**).

Figure 10.3

A CRC model index card



Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

These classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- *Controller classes* manage a "unit of work" from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

Quote:

"Objects can be classified scientifically into three major categories: those that don't work, those that break down, and those that get lost."

Russell Baker

What guidelines can be applied for allocating responsibilities to classes?

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 10.2 and 10.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. "Dumb" classes (those that have few responsibilities) can be modeled to act as servants to a few "smart" classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort.

   If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.[3] This enhances the maintainability of the software and reduces the impact of side effects due to change.

3 Cohesiveness is a design concept that is discussed in Chapter 12.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.[4] In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks*. The first operation (responsibility) implies a reasonably complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry,** a class that is encompassed by the aggregate class **CheckingAccount.**

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.** Each of these classes has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update* and *display* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

4 In such cases, it may be necessary to split the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes. Wirfs-Brock and her colleagues [Wir90] define collaborations in the following way:

> Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a status attribute to "not ready." Sensor information can be acquired from each **Sensor** object. Therefore, the responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor.**

To help in the identification of collaborators, you can examine three different generic relationships between classes [Wir90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

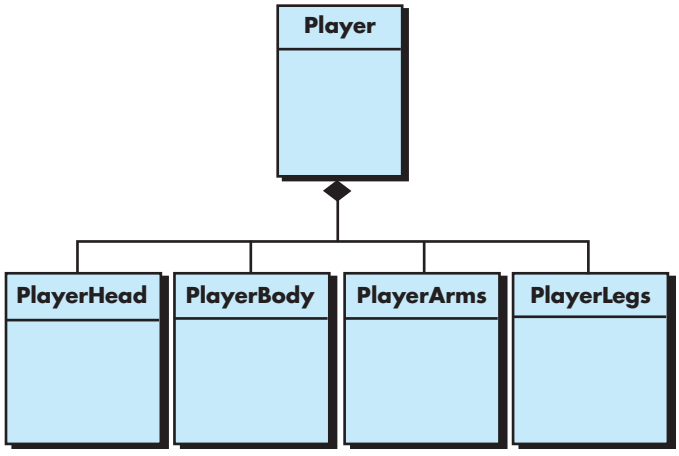
All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player**, as are **PlayerArms, PlayerLegs,** and **PlayerHead.** In UML, these relationships are represented as the aggregation shown in Figure 10.4.

When one class must acquire information from another class, the *has-knowledge-of* relationship is established. The *determine-sensor-status()* responsibility noted earlier is an example of a *has-knowledge-of* relationship.

The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of*. For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **PlayerHead** object called center-position is determined from the center position of **PlayerBody.** This information is obtained via a third object, **Player,** that acquires it from **PlayerBody.** Hence, **PlayerHead** *depends-upon* **PlayerBody.**

FIGURE 10.4

A composite aggregate class



In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 10.3).

When a complete CRC model has been developed, the representatives from the stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use case for *SafeHome* contains the following narrative:

> The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

When the review leader comes to "control panel," in the use case narrative, the token is passed to the person. holding the **ControlPanel** index card. The phrase "implies that a sensor is open" requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status()* accomplishes this). Next to the responsibility on the index card is the collaborator **Sensor.** The token is then passed to the **Sensor** object.

4. When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use case is finished. When all use cases (or use case diagrams) have been reviewed, requirements modeling continues.

SafeHome

CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:
[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

Vinod: While you've been working on surveillance and Jamie has been tied up with security, I've been working on the home management function.

Ed: What's the status of that? Marketing kept changing its mind.

Vinod: Here's the first-cut use case for the whole function . . . we've refined it a bit, but it should give you an overall view . . .

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a PC or an Internet connection to control electronic devices that have wireless interface controllers. The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, to set my heating and air-conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is home, another is away, a third is overnight travel, and a fourth is extended travel. All of these situations will have settings that will be applied to all devices. In the overnight travel and extended travel states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air-conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

Ed: The hardware guys have got all the wireless interfacing figured out?

Vinod (smiling): They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

Ed: Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

Vinod: I thought you didn't understand CRC.

Ed: Maybe a little, but go ahead.

Vinod: So here's my class definition for **HomeManagementInterface.**

Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

Operations:
displayControl(), selectControl(), displaySituation(), select situation(), accessFloorplan(), selectDeviceIcon(), displayDevicePanel(), accessDevicePanel(), . . .

Class: HomeManagementInterface

Responsibility	Collaborator
displayControl()	**OptionsPanel** (class)
selectControl()	**OptionsPanel** (class)
displaySituation()	**SituationPanel** (class)
selectSituation()	**SituationPanel** (class)
accessFloorplan()	**FloorPlan** (class) . . .
. . .	

Ed: So when the operation *accessFloorplan()* is invoked, it collaborates with the **FloorPlan** object just like the one we developed for surveillance. Wait, I have a description of it here. (They look at Figure 10.2.)

Vinod: Exactly. And if we wanted to review the entire class model, we could start with this index card, then go to the collaborator's index card, and from there to one of the collaborator's collaborators, and so on.

Ed: Good way to find omissions or errors.

Vinod: Yep.

10.5 Associations and Dependencies

KEY POINT

An association defines a relationship between classes. Multiplicity defines how many of one class are related to how many of another class.

In many instances, two analysis classes are related to one another in some fashion. In UML these relationships are called *associations*. Referring back to Figure 10.2, the **FloorPlan** class is defined by identifying a set of associations between **FloorPlan** and two other classes, **Camera** and **Wall**. The class **Wall** is associated with three classes that allow a wall to be constructed, **WallSegment**, **Window**, and **Door**.

In some cases, an association may be further defined by indicating *multiplicity*. Referring to Figure 10.2, a **Wall** object is constructed from one or more **WallSegment** objects. In addition, the **Wall** object may contain 0 or more **Window** objects and 0 or more **Door** objects. These multiplicity constraints are illustrated in Figure 10.5, where "one or more" is represented using 1..*, and "0 or more" by 0..*. In UML, the asterisk indicates an unlimited upper bound on the range.[5]

What is a stereotype?

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a *dependency relationship* is established. Dependencies are defined by a stereotype. A *stereotype* is an "extensibility mechanism" [Arl02] within UML that allows you to define a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., <<stereotype>>).

As an illustration of a simple dependency within the *SafeHome* surveillance system, a **Camera** object (in this case, the server class) provides a video image to

5 Other multiplicity relations—one to one, one to many, many to many, one to a specified range with lower and upper limits, and others—may be indicated as part of an association.

FIGURE 10.5
Multiplicity

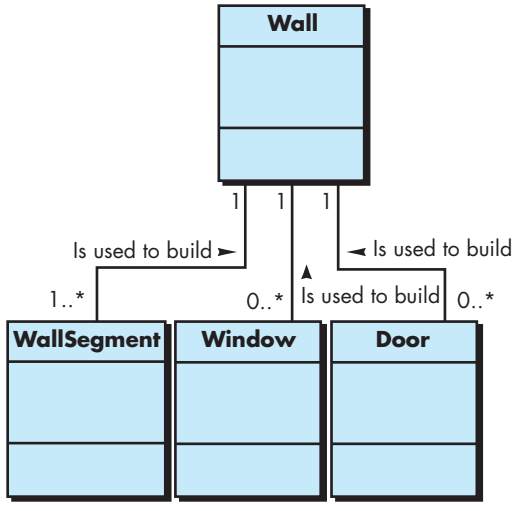
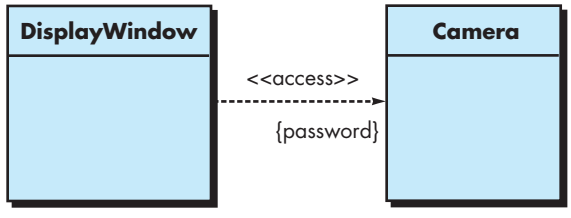


FIGURE 10.6
Dependencies



a **DisplayWindow** object (in this case, the client class). The relationship between these two objects is not a simple association, yet a dependency association does exist. In a use case written for surveillance (not shown), you learn that a special password must be provided in order to view specific camera locations. One way to achieve this is to have **Camera** request a password and then grant permission to the **DisplayWindow** to produce the video display. This can be represented as shown in Figure 10.6 where <<access>> implies that the use of the camera output is controlled by a special password.

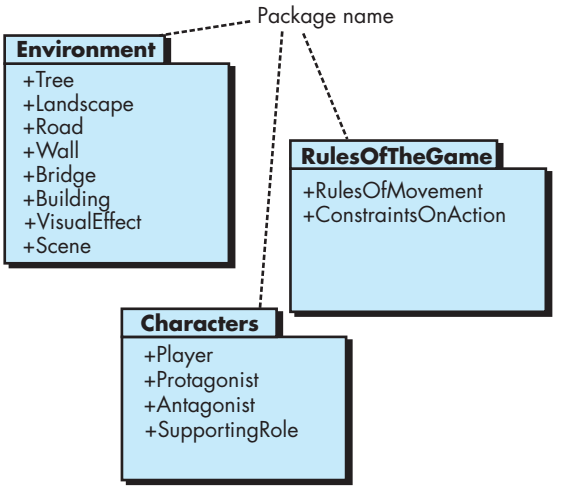
10.6 ANALYSIS PACKAGES

KEY POINT
A package is used to assemble a collection of related classes.

An important part of analysis modeling is categorization. That is, various elements of the requirements model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that we introduced earlier. As the analysis model for the video game is developed, a

FIGURE 10.7
Packages



large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as **Tree, Landscape, Road, Wall, Bridge, Building,** and **VisualEffect** might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints. Classes such as **Player** (described earlier), **Protagonist, Antagonist,** and **SupportingRoles** might be defined. Still others describe the rules of the game—how a player navigates through the environment. Classes such as **RulesOfMovement** and **ConstraintsOnAction** are candidates here. Many other categories might exist. These classes can be represented as analysis classes as shown in Figure 10.7.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

10.7 SUMMARY

Class-based modeling uses information derived from use cases and other written application descriptions to identify analysis classes. A grammatical parse may be used to extract candidate classes, attributes, and operations from text-based narratives. Criteria for the definition of a class are defined.

A set of class-responsibility-collaborator index cards can be used to define relationships between classes. In addition, a variety of UML modeling notation can be applied to define hierarchies, relationships, associations, aggregations,

and dependencies among classes. Analysis packages are used to categorize and group classes in a manner that makes them more manageable for large systems.

Problems and Points to Ponder

10.1. You have been asked to build one of the following systems:

a. A network-based course registration system for your university.
b. A Web-based order-processing system for a computer store.
c. A simple invoicing system for a small business.
d. An Internet-based cookbook that is built into an electric range or microwave.
Select the system that is of interest to you and develop a processing narrative. Then use the grammatical parsing technique to identify candidate objects and classes.

10.2. Develop a set of operations that are used within the classes identified in Problem 10.1.

10.3. Develop a class model for the PHTRS system present in Problem 9.5.

10.4. Write a template-based use case for the *SafeHome* home management system described informally in the sidebar following Section 10.4.

10.5. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 10.1.

10.6. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

10.7. What is an analysis package and how might it be used?

Further Readings and Information Sources

General class-based concepts are discussed by Weisfeld (*The Object-Oriented Thought Process*, 4th ed., Addison-Wesley, 2013). Class-based modeling methods are discussed in books by Bennet and Farmer (Object-Oriented Systems Analysis and Design Using UML, McGraw-Hill, 2010), Ashrafi and Ashrafi (*Object-Oriented Systems Analysis and Design,* Prentice Hall, 2008), Booch (*Object-Oriented Analysis and Design with Applications,* 3rd ed., Addison-Wesley, 2007), George and his colleagues (*Object-Oriented Systems Analysis and Design*, 2nd ed., Prentice Hall, 2006), O'Docherty (*Object-Oriented Analysis and Design,* Wiley, 2005), Satzinger et al. (*Object-Oriented Analysis and Design with the Unified Process,* Course Technology, 2004), Stumpf and Teague (*Object-Oriented Systems Analysis and Design with UML,* Prentice Hall, 2004).

UML modeling techniques that can be applied for both analysis and design are discussed by Dennis and his colleagues (*Systems Analysis and Design with UML Version 2.0,* Wiley, 4th ed., 2012), Ramnath and Dathan (*Object-Oriented Analysis and Design,* Springer, 2011), Bennett and Farmer (*Object-Oriented Systems Analysis and Design Using UML,* McGraw-Hill, 4th ed., 2010). Larman (*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* Dohrling Kindersley, 2008), Rosenberg and Stephens (*Use Case Driven Object Modeling with UML Theory and Practice,* Apress, 2007), and Arlow and Neustadt (*UML 2 and the Unified Process,* 2nd ed., Addison-Wesley, 2005) all address the definition of analysis classes within the context of UML.

A wide variety of information sources on class-based methods for requirements modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

11 Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps

Key Concepts

analysis patterns . . 208
behavioral model. . 203
configuration models 219
content model. . . . 216
events 203
functional model . . 218
interaction model . . 217
navigation modeling 220
sequence diagrams 205
state diagrams . . . 204
state representations 204

After our discussion of scenario-based and class-based models in Chapters 9 and 10, it's reasonable to ask, "Aren't those requirement modeling representations enough?"

The only reasonable answer is, "That depends."

For some types of software, the use case may be the only requirements modeling representation that is required. For others, an object-oriented approach is chosen and class-based models may be developed. But in other situations, complex application requirements may demand an examination of how an application behaves as a consequence of external events; whether existing domain knowledge can be adapted to the current problem; or in the case of Web-based or mobile systems and applications, how content and functionality meld to provide an end user with the ability to successfully navigate an application to achieve usage goals.

Quick Look

What is it? In this chapter you'll learn about other dimensions of the requirements model—behavioral models, patterns, and the special requirements analysis considerations that come into play when WebApps are developed. Each of these modeling representations supplements the scenario-based and class-based models discussed in Chapters 9 and 10.

Who does it? A software engineer (sometimes called an analyst) builds the model using requirements elicited from various stakeholders.

Why is it important? Your insight into software requirements grows in direct proportion to the number of different requirements modeling dimensions. Although you may not have the time, the resources, or the inclination to develop every representation suggested in Chapters 9 to 11, recognize that each different modeling approach provides you with a different way of looking at the problem. As a consequence, you (and other stakeholders) will be better able to assess whether you've properly specified what must be accomplished.

What are the steps? Behavioral modeling depicts the states of the system and its classes and the impact of events on these states. Pattern-based modeling makes use of existing domain knowledge to facilitate requirements analysis. WebApp requirements models are especially adapted for the representation of content, interaction, function, and configuration-related requirements.

What is the work product? A wide array of text-based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

11.1 Creating a Behavioral Model

The modeling notation that has been discussed in the preceding chapters represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time.

? How do I model the software's reaction to some external event?

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a state diagram for the system, and (5) review the behavioral model to verify accuracy and consistency. Each of these steps is discussed in the sections that follow.

11.2 Identifying Events with the Use Case

In Chapter 9, you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. An event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, reconsider the use case for a portion of the *SafeHome* security function.

> The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,**[1] transmits an event to the object **ControlPanel.** The event might be called *password entered*. The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered*

1 In this example, we assume that each user (homeowner) that interacts with *SafeHome* has an identifying password and is therefore a legitimate object.

does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

11.3 State Representations

KEY POINT

The system has states that represent specific externally observable behavior; a class has states that represent its behavior as the system performs its functions.

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current status of all of an object's attributes. For example, the passive state of the class **Player** (in the video game application discussed in Chapter 10) would include the current position and orientation attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates magic wishes remaining). The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: *moving, at rest, injured, being cured, trapped, lost,* and so forth. An *event* (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

Two different behavioral representations are discussed in the paragraphs that follow. The first indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time.

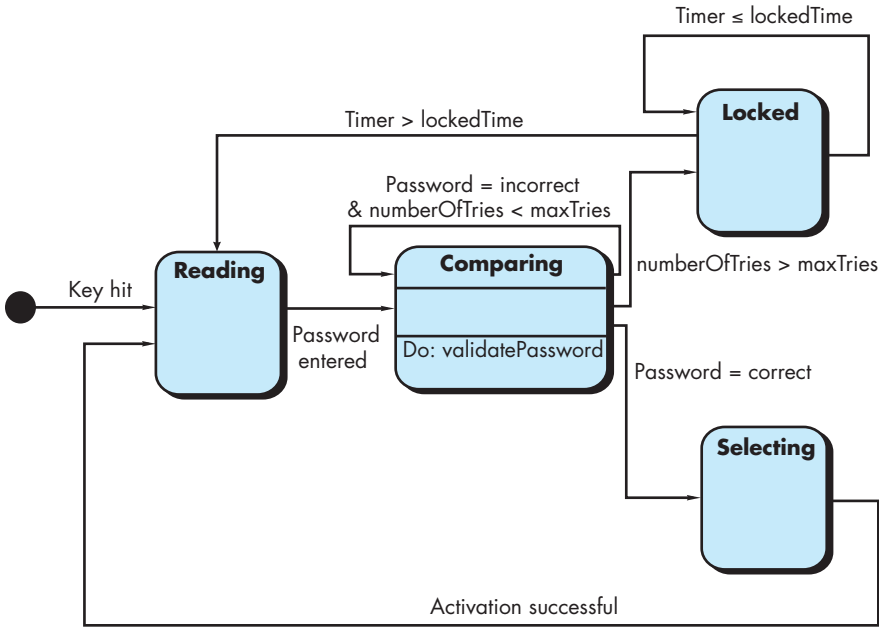
State Diagrams for Analysis Classes. One component of a behavioral model is a UML state diagram[2] that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 11.1 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

Each arrow shown in Figure 11.1 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to

2 If you are unfamiliar with UML, a brief introduction to this important modeling notation is presented in Appendix 1.

FIGURE 11.1

State diagram for the ControlPanel class



specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 11.1 can be determined by examining the use case:

if (password input = 4 digits) then *compare* to stored password

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

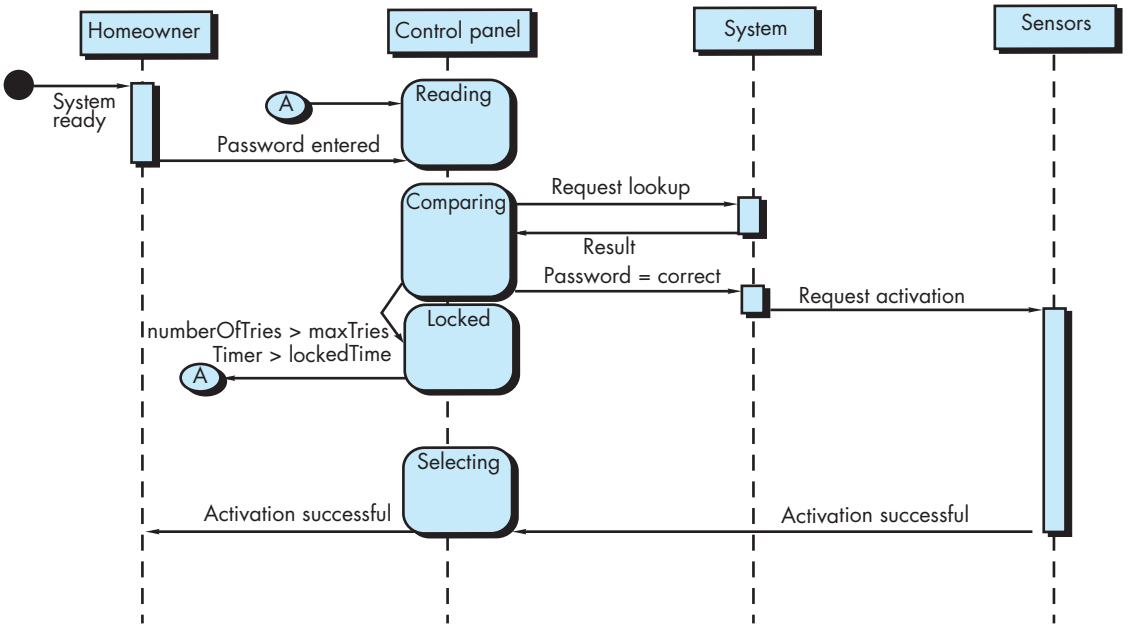
KEY POINT

Unlike a state diagram that represents behavior without noting the classes involved, a sequence diagram represents behavior, by describing how classes move from state to state.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 11.1) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

Sequence Diagrams. The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is

FIGURE 11.2 Sequence diagram (partial) for the *SafeHome* security function



a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 11.2 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System,** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System,** which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

SOFTWARE TOOLS

Generalized Analysis Modeling in UML

Objective: Analysis modeling tools provide the capability to develop scenario-based models, class-based models, and behavioral models using UML notation.

Mechanics: Tools in this category support the full range of UML diagrams required to build an analysis model (these tools also support design modeling). In addition to diagramming, tools in this category (1) perform consistency and correctness checks for all UML diagrams, (2) provide links for design and code generation, (3) build a database that enables the management and assessment of large UML models required for complex systems.

Representative Tools:[3]
The following tools support a full range of UML diagrams required for analysis modeling:

ArgoUML is an open source tool available at **argouml.tigris.org**.

Enterprise Architect, developed by Sparx Systems (**www.sparxsystems.com.au**).

PowerDesigner, developed by Sybase (**www.sybase.com**).

Rational Rose, developed by IBM (Rational) (**http://www-01.ibm.com/software/rational/**).

Rational System Architect, developed by Popkin Software now owned by IBM (**http://www-01.ibm.com/software/awdtools/systemarchitect/**).

UML Studio, developed by Pragsoft Corporation (**www.pragsoft.com**).

Visio, developed by Microsoft (**http://office.microsoft.com/en-gb/visio/**).

Visual UML, developed by Visual Object Modelers (**www.visualuml.com**).

11.4 PATTERNS FOR REQUIREMENTS MODELING

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. In other cases, the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

The original author of an analysis pattern does not "create" the pattern, but, rather, *discovers* it as requirements engineering work is being conducted. Once the pattern has been discovered, it is documented by describing "explicitly the general problem to which the pattern is applicable, the prescribed solution, assumptions and constraints of using the pattern in practice, and often some other information about the pattern, such as the motivation and driving forces for using the pattern, discussion of the pattern's advantages and disadvantages, and references to some known examples of using that pattern in practical applications" [Dev01].

In Chapter 8, we introduced the concept of analysis patterns and indicated that these patterns represent something (e.g., a class, a function, a behavior)

3 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

within the application domain that can be reused when performing requirements modeling for an application within a domain.[4] Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

11.4.1 Discovering Analysis Patterns

The requirements model comprises a wide variety of elements: scenario-based (use cases), class-based (objects and classes), and behavioral (events and states). Each of these elements represents the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the use case. In the context of this discussion, a coherent set of use cases may serve as the basis for discovering one or more analysis patterns. A *semantic analysis pattern* (SAP) "is a pattern that describes a small set of coherent use cases that together describe a basic generic application" [Fer00].

Consider the following preliminary use case for software required to control and monitor a real-view camera and proximity sensor for an automobile:

> **Use case:** *Monitor reverse motion*
>
> **Description:** When the vehicle is placed in *reverse* gear, the control software enables a video feed from a rear-placed video camera to the dashboard display. The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse. The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle. It will automatically brake the vehicle if the proximity sensor indicates an object within x feet of the rear of the vehicle, where x is determined based on the speed of the vehicle.

This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling. Regardless of how much elaboration is accomplished, the use cases suggest a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system. In this case, the "sensors" provide information about proximity and video information. The "actuator" is the braking system of the vehicle (invoked if an object is close to the vehicle). But in a more general case, a widely applicable pattern is discovered.

4 An in-depth discussion of the use of patterns during software design is presented in Chapter 16.

SAFEHOME

Discovering an Analysis Pattern

The scene: A meeting room, during a team meeting.

The players: Jamie Lazar, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager

The conversation:

Doug: How are things going with modeling the requirements for the sensor network for the *SafeHome* project?

Jamie: Sensor work is a little new to me, but I think I'm getting a handle on it.

Doug: Is there anything we can do to help you with that?

Jamie: It would be a lot easier if I'd built a system like this before.

Doug: True.

Ed: I was thinking this is a situation where we might be able to find an analysis pattern that would help us model these requirements.

Doug: If we can find the right pattern, we'd avoid reinventing the wheel.

Jamie: That sounds good to me. How do we start?

Ed: We have access to a repository that contains a large number of analysis and design patterns. We just need to search for patterns with intents that match our needs.

Doug: That seems like that might work. What do you think, Jamie?

Jamie: If Ed can help me get started, I'll tackle this today.

Software in many different application domains is required to monitor sensors and control physical actuators. It follows that an analysis pattern that describes generic requirements for this capability could be used widely. The pattern, called **Actuator-Sensor**, would be applicable as part of the requirements model for *SafeHome* and is discussed in Section 11.4.2.

11.4.2 A Requirements Pattern Example: Actuator-Sensor[5]

One of the requirements of the *SafeHome* security function is the ability to monitory security sensors (e.g., break-in sensors, fire, smoke or CO sensors, water sensors). Internet-based extensions to *SafeHome* will require the ability to control the movement (e.g., pan, zoom) of a security camera within a residence. The implication—*SafeHome* software must manage various sensors and "actuators" (e.g., camera control mechanisms).

Konrad and Cheng [Kon02] have suggested a requirements pattern named **Actuator-Sensor** that provides useful guidance for modeling this requirement within *SafeHome* software. An abbreviated version of the **Actuator-Sensor** pattern, originally developed for automotive applications, follows.

Pattern Name. **Actuator-Sensor**

Intent. Specify various kinds of sensors and actuators in an embedded system.

5 This section has been adapted from [Kon02] with the permission of the authors.

Motivation. Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The **Actuator-Sensor** pattern uses a *pull* mechanism (explicit request for information) for **PassiveSensors** and a *push* mechanism (broadcast of information) for the **ActiveSensors.**

Constraints

- Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
- Each active sensor must have capabilities to broadcast update messages when its value changes.
- Each active sensor should send a *life tick*, a status message issued within a specified time frame, to detect malfunctions.
- Each actuator must have some method to invoke the appropriate response determined by the **ComputingComponent.**
- Each sensor and actuator should have a function implemented to check its own operation state.
- Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

Applicability. Useful in any system in which multiple sensors and actuators are present.

Structure. UML class diagram for the **Actuator-Sensor** pattern is shown in Figure 11.3. **Actuator, PassiveSensor,** and **ActiveSensor** are abstract classes and denoted in italics. There are four different types of sensors and actuators in this pattern. The **Boolean, Integer,** and **Real** classes represent the most common types of sensors and actuators. The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device. Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

Behavior. Figure 11.4 presents a UML sequence diagram for an example of the **Actuator-Sensor** pattern as it might be applied for the *SafeHome* function that controls the positioning (e.g., pan, zoom) of a security camera. Here, the **ControlPanel**[6] queries a sensor (a passive position sensor) and an actuator (pan

6 The original pattern uses the generic phrase **ComputingComponent.**

FIGURE 11.3

UML sequence diagram for the Actuator-Sensor pattern. *Source:* Adapted from [Kon02] with permission.

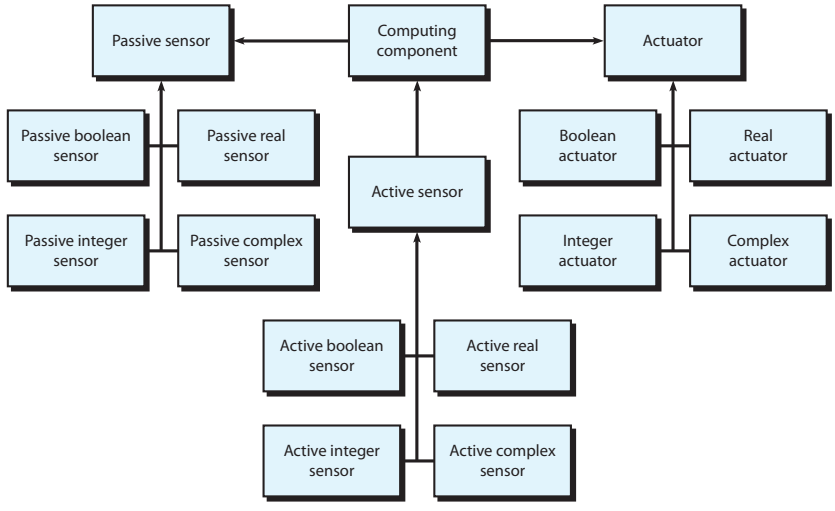
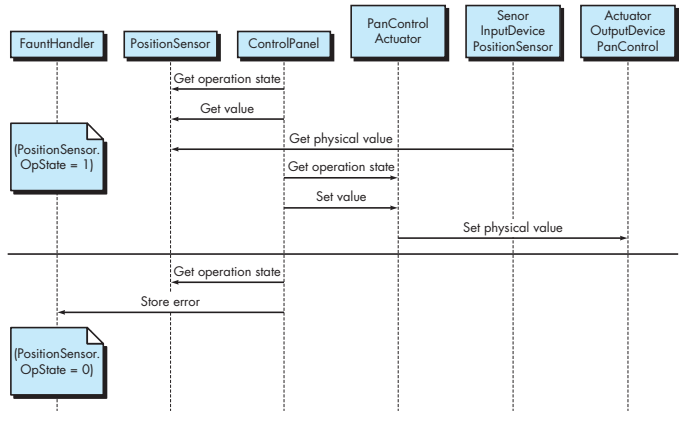


FIGURE 11.4

UML Class diagram for the Actuator-Sensor pattern. *Source:* Reprinted from [Kon02] with permission.



control) to check the operation state for diagnostic purposes before reading or setting a value. The messages *Set Physical Value* and *Get Physical Value* are not messages between objects. Instead, they describe the interaction between the physical devices of the system and their software counterparts. In the lower part of the diagram, below the horizontal line, the **PositionSensor** reports that the operation state is zero. The **ComputingComponent** then sends the error code for a position sensor failure to the **FaultHandler** that will decide how this error affects the system and what actions are required. It gets the data from the sensors and computes the required response for the actuators.

Participants. This section of the patterns description "itemizes the classes/objects that are included in the requirements pattern" [Kon02] and describes the responsibilities of each class/object (Figure 11.3). An abbreviated list follows:

- **PassiveSensor abstract:** Defines an interface for passive sensors.
- **PassiveBooleanSensor:** Defines passive Boolean sensors.
- **PassiveIntegerSensor:** Defines passive integer sensors.
- **PassiveRealSensor:** Defines passive real sensors.
- **ActiveSensor abstract:** Defines an interface for active sensors.
- **ActiveBooleanSensor:** Defines active Boolean sensors.
- **ActiveIntegerSensor:** Defines active integer sensors.
- **ActiveRealSensor:** Defines active real sensors.
- **Actuator abstract:** Defines an interface for actuators.
- **BooleanActuator:** Defines Boolean actuators.
- **IntegerActuator:** Defines integer actuators.
- **RealActuator:** Defines real actuators.
- **ComputingComponent:** The central part of the controller; it gets the data from the sensors and computes the required response for the actuators.
- **ActiveComplexSensor:** Complex active sensors have the basic functionality of the abstract **ActiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **PassiveComplexSensor:** Complex passive sensors have the basic functionality of the abstract **PassiveSensor** class, but additional, more elaborate, methods and attributes need to be specified.
- **ComplexActuator:** Complex actuators also have the base functionality of the abstract **Actuator** class, but additional, more elaborate methods and attributes need to be specified.

Collaborations. This section describes how objects and classes interact with one another and how each carries out its responsibilities.

- When the **ComputingComponent** needs to update the value of a **PassiveSensor,** it queries the sensors, requesting the value by sending the appropriate message.
- **ActiveSensors** are not queried. They initiate the transmission of sensor values to the computing unit, using the appropriate method to set the value in the **ComputingComponent.** They send a life tick at least once during a specified time frame in order to update their timestamps with the system clock's time.

- When the **ComputingComponent** needs to set the value of an actuator, it sends the value to the actuator.
- The **ComputingComponent** can query and set the operation state of the sensors and actuators using the appropriate methods. If an operation state is found to be zero, then the error is sent to the **FaultHandler,** a class that contains methods for handling error messages, such as starting a more elaborate recovery mechanism or a backup device. If no recovery is possible, then the system can only use the last known value for the sensor or the default value.
- The **ActiveSensors** offer methods to add or remove the addresses or address ranges of the components that want to receive the messages in case of a value change.

Consequences

1. Sensor and actuator classes have a common interface.
2. Class attributes can only be accessed through messages, and the class decides whether or not to accept the message. For example, if a value of an actuator is set above a maximum value, then the actuator class may not accept the message, or it might use a default maximum value.
3. The complexity of the system is potentially reduced because of the uniformity of interfaces for actuators and sensors.

The requirements pattern description might also provide references to other related requirements and design patterns.

11.5 Requirements Modeling for Web and Mobile Apps[7]

Developers of Web and mobile applications are often skeptical when the idea of requirements analysis is suggested. "After all," they argue, "our development process must be agile, and analysis is time consuming. It'll slow us down just when we need to be designing and building the application."

Requirements analysis does take time, but solving the wrong problem takes even more time. The question for every WebApp and mobile developer is simple—are you sure you understand the requirements of the problem or product? If the answer is an unequivocal yes, then it may be possible to skip requirements modeling, but if the answer is no, then requirements modeling should be performed.

7 Portions of this section has been adapted from Pressman and Lowe [Pre08] with permission.

11.5.1 How Much Analysis Is Enough?

The degree to which requirements modeling for Web and mobile apps is emphasized depends on the following size-related factors: (1) the size and complexity of the application increment, (2) the number of stakeholders (analysis can help to identify conflicting requirements coming from different sources), (3) the size of the app development team, (4) the degree to which members of the team have worked together before (analysis can help develop a common understanding of the project), and (5) the degree to which the organization's success is directly dependent on the success of the application.

The converse of the preceding points is that as the project becomes smaller, the number of stakeholders fewer, the development team more cohesive, and the application less critical, it is reasonable to apply a more lightweight analysis approach.

Although it is a good idea to analyze the problem or product requirements *before* beginning design, it is not true that *all* analysis must precede *all* design. In fact, the design of a specific part of the application only demands an analysis of those requirements that affect only that part of the application. As an example from *SafeHome*, you could validly design the overall website aesthetics (layouts, color schemes, etc.) without having analyzed the functional requirements for e-commerce capabilities. You only need to analyze that part of the problem that is relevant to the design work for the increment to be delivered.[8]

11.5.2 Requirements Modeling Input

An agile version of the generic software process discussed in Chapter 5 can be applied when Web or mobile apps are engineered. The process incorporates a communication activity that identifies stakeholders and user categories, the business context, defined informational and applicative goals, general product requirements, and usage scenarios—information that becomes input to requirements modeling. This information is represented in the form of natural language descriptions, rough outlines, sketches, and other informal representations.

Analysis takes this information, structures it using a formally defined representation scheme (where appropriate), and then produces more rigorous models as an output. The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution.

The *SafeHome* ACS-DCV (camera surveillance) function was introduced in Chapter 9. When it was introduced, this function seemed relatively clear and was described in some detail as part of a use case (Section 9.2.1). However, a reexamination of the use case might uncover information that is missing, ambiguous, or unclear.

Some aspects of this missing information would naturally emerge during the design. Examples might include the specific layout of the function buttons, their

8 In situations in which a design of one part of an application will have impact across other parts of an application, the scope of analysis should be broadened.

aesthetic look and feel, the size of snapshot views, the placement of camera views and the house floor plan, or even minutiae such as the maximum and minimum length of passwords. Some of these aspects are design decisions (such as the layout of the buttons) and others are requirements (such as the length of the passwords) that don't fundamentally influence early design work.

But some missing information might actually influence the overall design itself and relate more to an actual understanding of the requirements. For example:

Q1: What output video resolution is provided by *SafeHome* cameras?

Q2: What occurs if an alarm condition is encountered while the camera is being monitored?

Q3: How does the system handle cameras that can be panned and zoomed?

Q4: What information should be provided along with the camera view? (For example, location? time/date? last previous access?)

None of these questions were identified or considered in the initial development of the use case, and yet, the answers could have a substantial effect on different aspects of the design.

Therefore, it is reasonable to conclude that although the communication activity provides a good foundation for understanding, requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model, questions invariably arise. It is these questions that fill in the gaps—or in some cases, actually help us to find the gaps in the first place.

To summarize, the inputs to the requirements model will be the information collected during the communication activity—anything from an informal e-mail to a detailed project brief complete with comprehensive usage scenarios and product specifications.

11.5.3 Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating application content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp or mobile app resides.

Each of these characteristics can be represented as a set of models that allow application requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the application, there are five main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the application. Content includes text, graphics and images, video, and audio data.

- **Interaction model**—describes the manner in which users interact with the app.
- **Functional model**—defines the operations that will be applied to manipulate content and describes other processing functions that are independent of content but necessary to the end user.
- **Navigation model**—defines the overall navigation strategy for the app.
- **Configuration model**—describes the environment and infrastructure in which the app resides.

You can develop each of these models using a representation scheme (often called a "language") that allows its intent and structure to be communicated and evaluated easily among members of the engineering team and other stakeholders. As a consequence, a list of key issues (e.g., errors, omissions, inconsistencies, suggestions for enhancement or modification, points of clarification) are identified and acted upon.

11.5.4 Content Model

The content model contains structural elements that provide an important view of content requirements for an application. These structural elements encompass content objects and all *analysis classes*—user-visible entities that are created or manipulated as a user interacts with the app through a browser or a mobile device.[9]

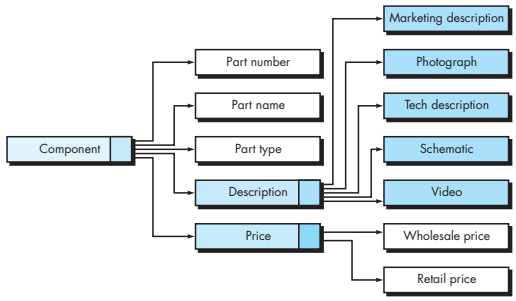
Content can be developed prior to the implementation of the app, while the app is being built, or long after the app is operational. In every case, it is incorporated via navigational reference into the overall application structure. A *content object* might be a textual description of a product, an article describing a news event, a graphical representation of retrieved data (e.g., stock price as a function of time), an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored as separate files or obtained dynamically from a database. They might be embedded directly into Web pages, displayed on the screen of a mobile device. In other words, a content object is any item of cohesive information that is to be presented to an end user.

Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. For example, a WebApp that supports *SafeHome* is established at **www.safehomeassured.com.** A use case, *Purchasing Select SafeHome Components,* describes the scenario required to purchase a *SafeHome* component and contains the sentence:

> I will be able to get descriptive and pricing information for each product component.

9 Analysis classes were discussed in Chapter 10.

FIGURE 11.5 Data tree for a **www.safehomeassured.com** component



The content model must be capable of describing the content object **Component.** In many instances, a simple list of content objects, coupled with a brief description of each object, is sufficient to define the requirements for content that must be designed and implemented. However, in some cases, the content model may benefit from a richer analysis that graphically illustrates the relationships among content objects and/or the hierarchy of content maintained by a WebApp.

For example, consider the *data tree* [Sri01] created for a **www.safehomeassured.com** component shown in Figure 11.5. The tree represents a hierarchy of information that is used to describe a component. Simple or composite data items (one or more data values) are represented as unshaded rectangles. Content objects are represented as shaded rectangles. In the figure, description is defined by five content objects (the shaded rectangles). In some cases, one or more of these objects would be further refined as the data tree expands.

A data tree can be created for any content that is composed of multiple content objects and data items. The data tree is developed in an effort to define hierarchical relationships among content objects and to provide a means for reviewing content so that omissions and inconsistencies are uncovered before design commences. In addition, the data tree serves as the basis for content design.

11.5.5 Interaction Model for Web and Mobile Apps

The vast majority of Web and mobile apps enable a "conversation" between an end user and application functionality, content, and behavior. This conversation can be described using an *interaction* model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,[10] and/or (4) user interface prototypes.

In many instances, a set of use cases[11] is sufficient to describe the interaction at an analysis level (further refinement and detail is introduced during design).

10 Sequence diagrams and state diagrams are modeled using UML notation.

11 Use cases are described in detail in Chapter 9.

However, when the sequence of interaction is complex and involves multiple analysis classes or many tasks, it is sometimes worthwhile to depict it using a more rigorous diagrammatic form.

The layout of the user interface, the content it presents, the interaction mechanisms it implements, and the overall aesthetic of the user to app connection have much to do with user satisfaction and the overall success of the app. Although it can be argued that the creation of a user interface prototype is a design activity, it is a good idea to perform it during the creation of the analysis model. The sooner that a physical representation of a user interface can be reviewed, the higher the likelihood that end users will get what they want. The design of user interfaces is discussed in detail in Chapter 15.

Because Web and mobile app construction tools are plentiful, relatively inexpensive, and functionally powerful, it is best to create the interface prototype using such tools. The prototype should implement the major navigational links and represent the overall screen layout in much the same way that it will be constructed. For example, if five major system functions are to be provided to the end user, the prototype should represent them as the user will see them upon first entering the app. Will graphical links be provided? Where will the navigation menu be displayed? What other information will the user see? Questions like these should be answered by the prototype.

11.5.6 Functional Model

Many WebApps deliver a broad array of computational and manipulative functions that can be associated directly with content (either using it or producing it) and that are often a major goal of user-WebApp interaction. Mobile apps tend to be more focused and provide a more limited set of computational and manipulative functions. Regardless of the breadth of functionality, functional requirements should be analyzed, and when necessary, modeled.

The *functional model* addresses two app processing elements, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the app to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user. For example, a financial mobile app might implement a variety of financial functions (e.g., computation of mortgage payment). These functions may actually be implemented using operations within analysis classes, but from the point of view of the end user, the function (more correctly, the data provided by the function) is the visible outcome.

At a lower level of procedural abstraction, the requirements model describes the processing to be performed by analysis class operations. These operations manipulate class attributes and are involved as classes collaborate with one another to accomplish some required behavior.

Regardless of the level of procedural abstraction, the UML activity diagram can be used to represent processing details. At the analysis level, activity diagrams should be used only where the functionality is relatively complex. Much of the complexity of WebApps and mobile apps occurs not in the functionality provided, but rather with the nature of the information that can be accessed and the ways in which this can be manipulated.

An example of relatively complex functionality for **www.safehomeassured.com** is addressed by a use case entitled *Get recommendations for sensor layout for my space*. The user has already developed a layout for the space to be monitored, and in this use case, selects that layout and requests recommended locations for sensors within the layout. **www.safehomeassured.com** responds with a graphical representation of the layout with additional information on the recommended locations for sensors. The interaction is quite simple, the content is somewhat more complex, but the underlying functionality it very sophisticated. The system must undertake a relatively complex analysis of the floor layout in order to determine the optimal set of sensors. It must examine room dimensions, the location of doors and windows, and coordinate these with sensor capabilities and specifications. No small task! A set of activity diagrams can be used to describe processing for this use case.

The second example is the use case *Control cameras*. In this use case, the interaction is relatively simple, but there is the potential for complex functionality, given that this "simple" operation requires complex communication with devices located remotely and accessible across the Internet. A further possible complication relates to negotiation of control when multiple authorized people attempt to monitor and/or control a single sensor at the same time.

Figure 11.6 depicts an activity diagram for the *takeControlOfCamera()* operation that is part of the **Camera** analysis class used within the *Control cameras* use case. It should be noted that two additional operations are invoked with the procedural flow: *requestCameraLock(),* which tries to lock the camera for this user, and *getCurrentCameraUser(),* which retrieves the name of the user who is currently controlling the camera. The construction details indicating how these operations are invoked and the interface details for each operation are not considered until WebApp design commences.

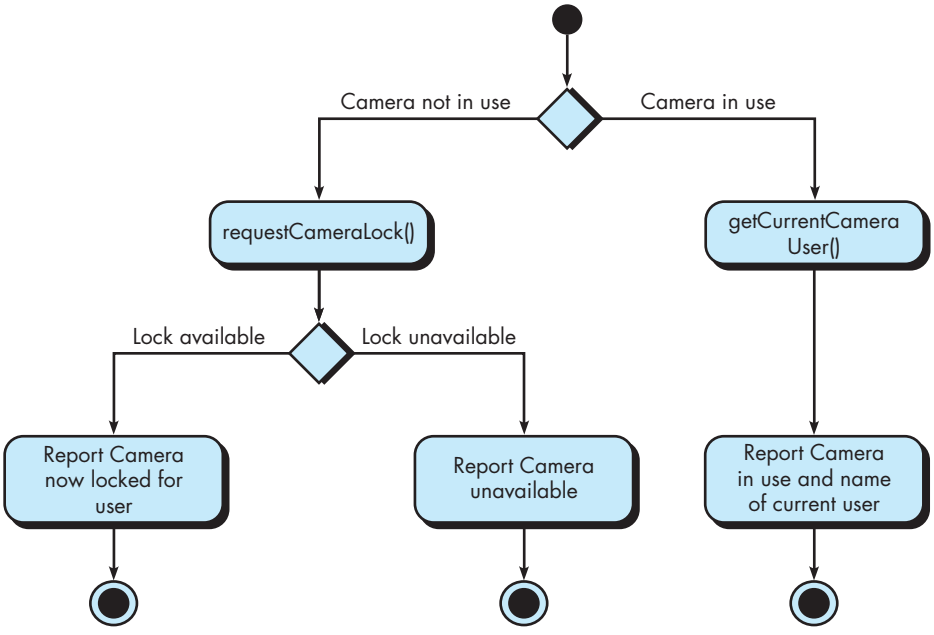
An extension of *SafeHome* WebApp functionality might occur with the development of a mobile app that provides access to the *SafeHome* system from a smart phone or tablet. The content and functional requirements for a *SafeHome* mobile app might be similar to a subset of those provided by the WebApp, but specific interface and security requirements would have to be established.

11.5.7 Configuration Models for WebApps

In some cases, the configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex apps, a variety of configuration

FIGURE 11.6

Activity diagram for the *takeControlOf-Camera()* operation



complexities (e.g., distributing load among multiple servers, caching architectures, remote databases, multiple servers serving various objects) may have an impact on analysis and design. The UML *deployment diagram* can be used in situations in which complex configuration architectures must be considered.

For **www.safehomeassured.com** the public content and functionality should be specified to be accessible across all major Web clients (i.e., those with more than 1 percent market share or greater). Conversely, it may be acceptable to restrict the more complex control and monitoring functionality (which is only accessible to **HomeOwner** users) to a smaller set of clients. For a mobile app, implementation might be limited to the three leading mobile operating environments. The configuration model for **www.safehomeassured.com** will also specify interoperability with existing product databases and monitoring applications.

11.5.8 Navigation Modeling

In most mobile applications that reside on smartphone platforms, navigation is generally constrained to relatively simple button lists and icon-based menus. In addition, the depth of navigation (i.e., the number of levels into the hypermedia hierarchy) is relatively shallow. For these reasons, navigation modeling is relatively simple.

For WebApps and an increasing number of tablet-based mobile applications, navigation modeling is more complex and often considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics

of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu (as opposed to a single "back" link or directed pointer) be available at every point in a user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "store" his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.

You and other stakeholders must also determine overall requirements for navigation. For example, will a "site map" be provided to give users an overview of the entire WebApp structure? Can a user take a "guided tour" that will highlight the most important elements (content objects and functions) that are available? Will a user be able to access content objects or functions based on defined attributes of those elements (e.g., a user might want to access all photographs of a specific building or all functions that allow computation of weight)?

11.6 Summary

Behavioral modeling during requirements analysis depicts dynamic behavior of the software. The behavioral model uses input from scenario-based or class-based elements to represent the states of analysis classes and the system as a

whole. To accomplish this, states are identified, the events that cause a class (or the system) to make a transition from one state to another are defined, and the actions that occur as transition is accomplished are also identified. State diagrams and sequence diagrams are the notation used for behavioral modeling.

Analysis patterns enable a software engineer to use existing domain knowledge to facilitate the creation of a requirements model. An analysis pattern describes a specific software feature or function that can be described by a coherent set of use cases. It specifies the intent of the pattern, the motivation for its use, constraints that limit its use, its applicability in various problem domains, the overall structure of the pattern, its behavior and collaborations, and other supplementary information.

Requirements modeling for mobile applications and WebApps can use most, if not all, of the modeling elements discussed in this book. However, these elements are applied within a set of specialized models that address content, interaction, function, navigation, and the configuration in which the mobile app or WebApp resides.

Problems and Points to Ponder

11.1. There are two different types of "states" that behavioral models can represent. What are they?

11.2. How does a sequence diagram differ from a state diagram? How are they similar?

11.3. Suggest three requirements patterns for a modern mobile phone and write a brief description of each. Could these patterns be used for other devices? Provide an example.

11.4. Select one of the patterns you developed in Problem 11.3 and develop a reasonably complete pattern description similar in content and style to the one presented in Section 11.4.2.

11.5. How much analysis modeling do you think would be required for **www.safehomeassured.com**? Would each of the model types described in Section 11.5.3 be required?

11.6. What is the purpose of the interaction model for a WebApp?

11.7. It could be argued that a WebApp functional model should be delayed until design. Present pros and cons for this argument.

11.8. What is the purpose of a configuration model?

11.9. How does the navigation model differ from the interaction model?

Further Readings and Information Sources

Behavioral modeling presents an important dynamic view of system behavior. Books by Samek (*Practical UML Statecharts in C/C++: Event Driven Programming for Embedded Systems,* CRC Press, 2008), Wagner and his colleagues (*Modeling Software with Finite State Machines: A Practical Approach,* Auerbach, 2006) and Boerger and Staerk (*Abstract State Machines,* Springer, 2003) present thorough discussions of state diagrams and other behavioral representations. Gomes and Fernandez (*Behavioral Modeling for Embedded Systems*

and Technologies, Information Science Reference, 2009) have edited an anthology that addresses behavioral modeling techniques for embedded systems.

The majority of books written about software patterns focus on software design. However, books by Vaughn (*Implementing Domain-Driven Design*, Addison-Wesley, 2013), Whithall (*Software Requirement Patterns,* Microsoft Press, 2007), Evans (*Domain-Driven Design,* Addison-Wesley, 2003) and Fowler ([Fow03] and [Fow97]) address analysis patterns specifically.

An in-depth treatment of analysis modeling for WebApps is presented by Pressman and Lowe [Pre08]. Books by Rossi and his colleagues (*Web Engineering: Modeling and Implementing Web Applications,* Springer, 2010) and Neil (*Mobile Design Pattern Gallery: UI Patterns,* O'Reilly, 2012) discuss the use of patterns in app development. Papers contained within an anthology edited by Murugesan and Desphande *(Web Engineering: Managing Diversity and Complexity of Web Application Development,* Springer, 2001) treat various aspects of WebApp requirements. In addition, the annual *Proceeding of the International Conference on Web Engineering* regularly addresses requirements modeling issues.

A wide variety of information sources on requirements modeling are available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis modeling can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

12 Design Concepts

Key Concepts

abstraction 232
architecture 232
aspects. 237
cohesion 236
data design. 244
design process . . . 228
functional independence 236
good design 228
information hiding. 235
modularity 234
object-oriented design 238
patterns 233
quality attributes . . 230

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design is pivotal to successful software engineering. In the early 1990s Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal*. He wrote:

> What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together . . .

Quick Look

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks.

Why is it important? Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.

What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

What is the work product? A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design.

How do I ensure that I've done it right? The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

quality guidelines . . 228
refactoring 238
separation of concerns 234
software design . . 230
stepwise refinement 237

> The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software. *Firmness:* A program should not have any bugs that inhibit its function. *Commodity:* A program should be suitable for the purposes for which it was intended. *Delight:* The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you must practice diversification and then convergence. Belady [Bel81] states that "diversification is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind." Once this diverse set of information is assembled, you must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model (Chapters 8 to 11). As this occurs, alternatives are considered and rejected, and you converge on "one particular configuration of components, and thus the creation of the final product" [Bel81].

Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Quote:

"The most common miracle of software engineering is the transition from analysis to design and design to code."

Richard Due'

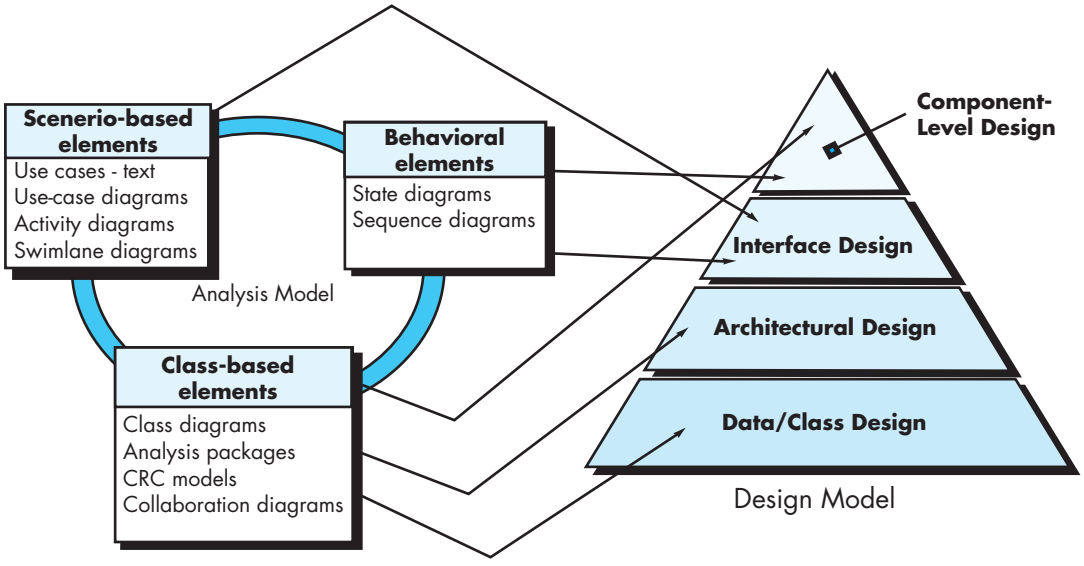
Software design changes continually as new methods, better analysis, and broader understanding evolve.[1] Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied. In this chapter, we explore the fundamental concepts and principles that are applicable to all software design, the elements of the design model, and the impact of patterns on the design process. In Chapters 12 to 18 we'll present a variety of software design methods as they are applied to architectural, interface, and component-level design as well as pattern-based and Web-oriented design approaches.

12.1 Design within the Context of Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last

1 Those readers with further interest in the philosophy of software design might have interest in Philippe Kruchen's intriguing discussion of "post-modern" design [Kru05].

FIGURE 12.1 Translating the requirements model into the design model



software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

Each of the elements of the requirements model (Chapters 9–11) provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 12.1. The requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

ADVICE

Software design should always begin with a consideration of data—the foundation for all other elements of the design. After the foundation is laid, the architecture must be derived. Only then should you perform other design tasks.

The data/class design transforms class models (Chapter 10) into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design activity. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns (Chapter 13 that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented [Sha96]. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

Quote:

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

SafeHome

Design versus Coding

The scene: Jamie's cubicle, as the team prepares to translate requirements into design.

The players: Jamie, Vinod, and Ed—all members of the *SafeHome* software engineering team.

The conversation:

Jamie: You know, Doug [the team manager] is obsessed with design. I gotta be honest, what I really love doing is coding. Give me C++ or Java, and I'm happy.

Ed: Nah . . . you like to design.

Jamie: You're not listening—coding is where it's at.

Vinod: I think what Ed means is you don't really like coding; you like to design and express it in code. Code is the language you use to represent the design.

Jamie: And what's wrong with that?

Vinod: Level of abstraction.

Jamie: Huh?

Ed: A programming language is good for representing details like data structures and algorithms, but it's not so good for representing architecture or component-to-component collaboration . . . stuff like that.

Vinod: And a screwed-up architecture can ruin even the best code.

Jamie (thinking for a minute): So, you're saying that I can't represent architecture in code . . . that's not true.

Vinod: You can certainly imply architecture in code, but in most programming languages, it's pretty difficult to get a quick, big-picture read on architecture by examining the code.

Ed: And that's what we want before we begin coding.

Jamie: Okay, maybe design and coding are different, but I still like coding better.

12.2 The Design Process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

12.2.1 Software Quality Guidelines and Attributes

Quote:

"[W]riting a clever piece of code that works is one thing; designing something that can support a long-lasting business is quite another."

C. Ferguson

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed in Chapter 20. McGlaughlin [McG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

- The design should implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

Quality Guidelines. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 12.3, we discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

? What are the characteristics of a good design?

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,[2] thereby facilitating implementation and testing.

2 For smaller systems, design can sometimes be developed linearly.

Quote:

"Design is not just what it looks like and feels like. Design is how it works."

Steve Jobs

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

INFO

Assessing Design Quality—The Technical Review

Design is important because it allows a software team to assess the quality[3] of the software before it is implemented—at a time when errors, omissions, or inconsistencies are easy and inexpensive to correct. But how do we assess quality during design? The software can't be tested, because there is no executable software to test. What to do?

During design, quality is assessed by conducting a series of technical reviews (TRs). TRs are discussed in detail in Chapter 20,[4] but it's worth providing a summary of the technique at this point. A technical review is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed. Each person plays a role: the *review leader* plans the meeting, sets an agenda, and runs the meeting; the *recorder* takes notes so that nothing is missed; the *producer* is the person whose work product (e.g., the design of a software component) is being reviewed. Prior to the meeting, each person on the review team is given a copy of the design work product and is asked to read it, looking for errors, omissions, or ambiguity. When the meeting commences, the intent is to note all problems with the work product so that they can be corrected before implementation begins. The TR typically lasts between 60 to 90 minutes. At the conclusion of the TR, the review team determines whether further actions are required on the part of the producer before the design work product can be approved as part of the final design model.

3 The quality factors discussed in Chapter 30 can assist the review team as it assesses quality.

4 You might consider looking ahead to Chapter 20 at this time. Technical reviews are a critical part of the design process and are an importance mechanism for achieving design quality.

Quote:

"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree."

Robert Pirsig

ADVICE

Software designers tend to focus on the problem to be solved. Just don't forget that the FURPS attributes are always part of the problem. They must be considered.

Quality Attributes. Hewlett-Packard [Gra87] developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors (Chapters 6 and 15), overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured using processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 29), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

12.2.2 The Evolution of Software Design

Quote:

"A designer knows that he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

Antoine de St-Expurey

The evolution of software design is a continuing process that has now spanned more than six decades. Early design work concentrated on criteria for the development of modular programs [Den73] and methods for refining software structures in a top-down "structured" manner ([Wir71], [Dah72], [Mil72]). Newer design approaches (e.g., [Jac92], [Gam95]) proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture [Kru06] and the design patterns that can be used to implement software architectures and lower levels of design abstractions (e.g., [Hol06], [Sha05]). Growing emphasis on aspect-oriented methods (e.g., [Cla05], [Jac04]), model-driven development [Sch06], and test-driven development [Ast04] emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

? What characteristics are common to all design methods?

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapters 9 to 11, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

TASK SET

Generic Task Set for Design

1. Examine the information domain model and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style (pattern) that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components:
   - Translate analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select design patterns for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create behavioral model of the interface.
   - Define interface objects, control mechanisms.
   - Review the interface design and revise as required.
7. Conduct component-level design.
   - Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.
   - Review each component and correct all errors uncovered.
8. Develop a deployment model.

12.3 DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in these concepts has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you define criteria that can be used to partition software

into individual components, separate or data structure detail from a conceptual representation of the software, and establish uniform criteria that define the technical quality of a software design.

M. A. Jackson [Jac75] once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right." In the sections that follow, we present an overview of fundamental software design concepts that provide the necessary framework for "getting it right."

Quote:

"Abstraction is one of the fundamental ways that we as humans cope with complexity."

Grady Booch

12.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

ADVICE

As a designer, work hard to derive both procedural and data abstractions that serve the problem at hand. If they can serve an entire domain of problems, that's even better.

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).[5]

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

WebRef

An in-depth discussion of software architecture can be found at **www.sei.cmu.edu/ata/ata_init.html**.

12.3.2 Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the

5 It should be noted, however, that one set of operations can be replaced with another, as long as the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *open* would change dramatically if the door were automatic and attached to a sensor.

structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Quote:

"A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule, and cost."

Len Bass et al.

Shaw and Garlan [Sha95a] describe a set of properties that should be specified as part of an architectural design. *Structural properties* define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." *Extra-functional properties* address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics. *Families of related systems* "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [Gar95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.

Quote:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

A number of different *architectural description languages* (ADLs) have been developed to represent these models [Sha95b]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

You should note that there is some debate about the role of architecture in design. Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process. The manner in which software architecture is characterized and its role in design are discussed in Chapter 13.

12.3.3 Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00].

Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern. Design patterns are discussed in detail in Chapter 16.

12.3.4 Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

It follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

12.3.5 Modularity

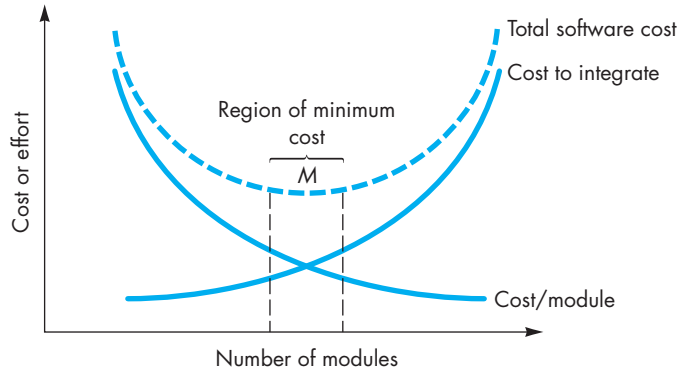
Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules,* that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 12.2, the effort (cost) to develop an individual software module does decrease as the total number of

FIGURE 12.2

Modularity and software cost



modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, *M,* of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.

? What is the right number of modules for a given system?

The curves shown in Figure 12.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of *M*. Undermodularity or overmodularity should be avoided. But how do you know the vicinity of *M*? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

12.3.6 Information Hiding

KEY POINT

The intent of information hiding is to hide the details of data structures and procedural processing behind a module interface. Knowledge of the details need not be known by users of the module.

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of *information hiding* [Par72] suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the

procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

12.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design Wirth [Wir71] and Parnas [Par72] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

? Why should you strive to create independent modules?

It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

KEY POINT

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in Section 12.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

KEY POINT

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across

the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [Ste74], caused when errors occur at one location and propagate throughout a system.

12.3.8 Refinement

ADVICE

There is a tendency to move immediately to full detail, skipping refinement steps. This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement.

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. An application is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of *elaboration*. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

12.3.9 Aspects

Quote:

"It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design."

Bruce Tognazzini

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts" [AOS07]. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account" [Ros04].

KEY POINT

A crosscutting concern is some characteristic of the system that applies across many different requirements.

For example, consider two requirements for the **www.safehomeassured.com** WebApp. Requirement *A* is described via the ACS-DCV use case discussed in Chapter 9. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **www.safehomeassured.com.** This requirement

is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, A^* is a design representation for requirement A and B^* is a design representation for requirement B. Therefore, A^* and B^* are representations of concerns, and B^* *crosscuts* A^*.

An *aspect* is a representation of a crosscutting concern. Therefore, the design representation, B^*, of the requirement *a registered user must be validated prior to using* **www.safehomeassured.com,** is an aspect of the *SafeHome* WebApp. It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components [Ban06a]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

12.3.10 Refactoring

WebRef
Excellent resources for refactoring can be found at **www.refactoring.com**.

An important design activity suggested for many agile methods (Chapter 5), *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

WebRef
A variety of refactoring patterns can be found at **http://c2.com/cgi/wiki?Refactoring Patterns**.

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. As a consequence, refactoring tools [Soa10] are used to analyze changes automatically and to "generate a test suite suitable for detecting behavioral changes."

12.3.11 Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. Appendix 2 has been provided for those readers who may be unfamiliar with OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

SAFEHOME

Design Concepts

The scene: Vinod's cubicle, as design modeling begins.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

The conversation:

[All four team members have just returned from a morning seminar entitled "Applying Basic Design Concepts," offered by a local computer science professor.]

Vinod: Did you get anything out of the seminar?

Ed: Knew most of the stuff, but it's not a bad idea to hear it again, I suppose.

Jamie: When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

Vinod: Because . . . bottom line . . . it's a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

Shakira: I wasn't a CS grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don't see why this stuff is so important.

Jamie: I've seen your work, Shak, and you know what, you do a lot of this stuff naturally . . . that's why your designs and code work.

Shakira (smiling): Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . that sort of thing.

Ed: Modularity, functional independence, hiding, patterns . . . see.

Jamie: I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

Vinod: Same thing can be applied to design, you know.

Jamie: The only concepts I hadn't heard of before were "aspects" and "refactoring."

Shakira: That's used in Extreme Programming, I think she said.

Ed: Yep. It's not a whole lot different than refinement, only you do it after the design or code is completed. Kind of an optimization pass through the software, if you ask me.

Jamie: Let's get back to *SafeHome* design. I think we should put these concepts on our review checklist as we develop the design model for *SafeHome*.

Vinod: I agree. But as important, let's all commit to think about them as we develop the design.

? What types of classes does the designer create?

12.3.12 Design Classes

The analysis model defines a set of analysis classes (Chapter 10). Each of these classes describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]. *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor. *Business domain classes* identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or

more analysis classes. *Process classes* implement lower-level business abstractions required to fully manage the business domain classes. *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software. *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class (Chapter 10) is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

? What is a "well-formed" design class?

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip** for video-editing software might have attributes **start point** and **end point** to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, *setStartPoint()* and *setEndPoint(),* provide the only means for establishing start and end points for the clip.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to

implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.[6]

SAFEHOME

Refining an Analysis Class into a Design Class

The scene: Ed's cubicle, as design modeling begins.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

[Ed is working on the **FloorPlan** class (see sidebar discussion in Section 10.3 and Figure 10.2) and has refined it for the design model.]

Ed: So you remember the **FloorPlan** class, right? It's used as part of the surveillance and home management functions.

Vinod (nodding): Yeah, I seem to recall that we used it as part of our CRC discussions for home management.

Ed: We did. Anyway, I'm refining it for design. Want to show how we'll actually implement the **FloorPlan** class. My idea is to implement it as a set of linked lists [a specific data structure]. So . . . I had to refine the analysis class **FloorPlan** (Figure 10.2) and actually, sort of simplify it.

Vinod: The analysis class showed only things in the problem domain, well, actually on the computer screen, that were visible to the end user, right?

Ed: Yep, but for the **FloorPlan** design class, I've got to add some things that are implementation specific. I needed to show that **FloorPlan** is an aggregation of segments—hence the **Segment** class—and that the **Segment** class is composed of lists for wall segments, windows, doors, and so on. The class **Camera** collaborates with **FloorPlan,** and obviously, there can be many cameras in the floor plan.

Vinod: Phew, let's see a picture of this new **FloorPlan** design class.

[Ed shows Vinod the drawing shown in Figure 12.3.]

Vinod: Okay, I see what you're trying to do. This allows you to modify the floor plan easily because new items can be added to or deleted from the list—the aggregation—without any problems.

Ed (nodding): Yeah, I think it'll work.

Vinod: So do I.

12.3.13 Dependency Inversion

The structure of many older software architectures is hierarchical. At the top of the architecture, "control" components rely on lower-level "worker" components to perform various cohesive tasks. Consider a simple program with three components. The intent of the program is to read keyboard strokes and then print the result to a printer. A control module, *C*, coordinates two other modules—a keystroke reader module, *R*, and a module that writes to a printer, *W*.

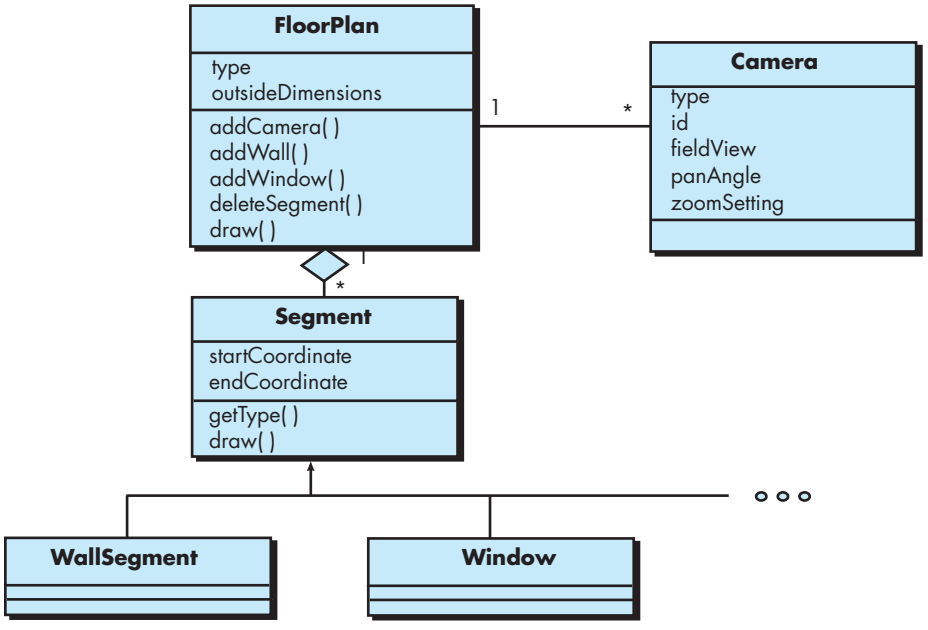
? What is the "dependency inversion principle"?

The design of the program is coupled because *C* is highly dependent on *R* and *W*. To remove the level of dependence that exists, the "worker" modules *R* and *W* should be invoked from the control module *S* using abstractions. In

6 A less formal way of stating the Law of Demeter is "Each unit should only talk to its friends; Don't talk to strangers."

FIGURE 12.3

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



object-oriented software engineering, abstractions are implemented as abstract classes, **R*** and **W*.** These abstract classes could then be used to invoke worker classes that perform any read and write function. Therefore a **copy** class, **C,** invokes abstract classes, **R*** and **W*,** and the abstract class points to the appropriate worker-class (e.g., the **R*** class might point to a *read()* operation within a **keyboard** class in one context and a *read()* operation within a **sensor** class in another. This approach reduces coupling and improves the testability of a design.

The example discussed in the preceding paragraph can be generalized with the *dependency inversion principle* [Obj10], which states: *High-level modules (classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

12.3.14 Design for Test

There is an ongoing chicken-and-egg debate about whether software design or test case design should come first. Rebecca Wirfs-Brock [Wir09] writes:

> Advocates of test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters' credo, "Test fast, fail fast, adjust fast." Testing guides their design as they implement in short, rapid-fire "write test code—fail the test—write enough code to pass—then pass the test" cycles.

Quote:

"Test fast, fail fast, adjust fast."

Tom Peters

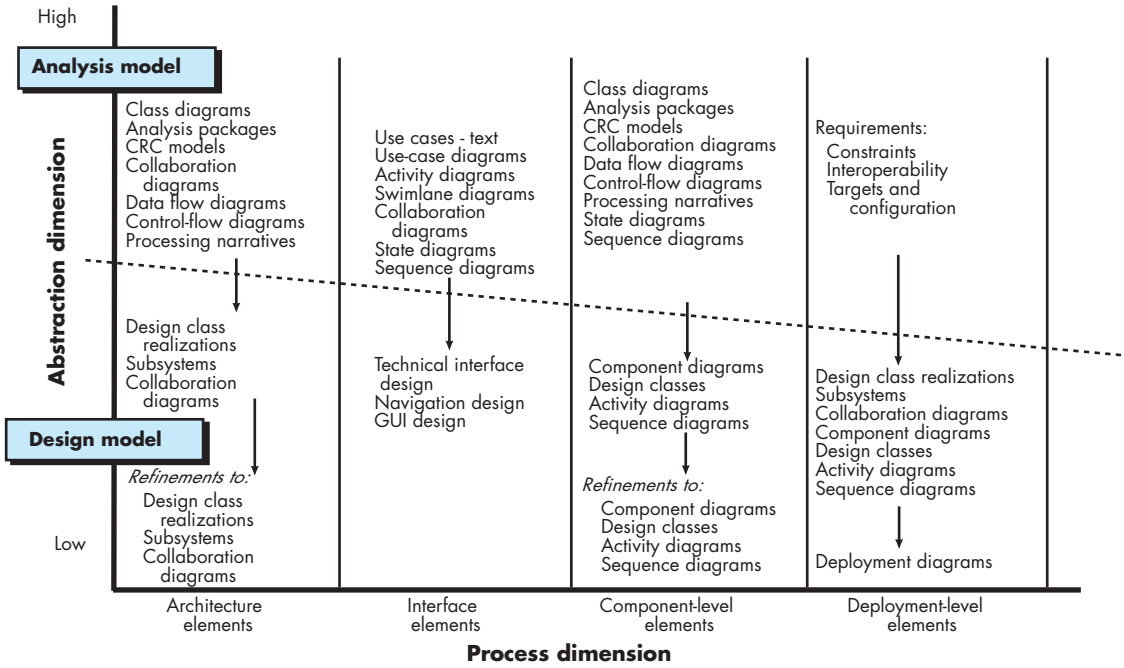
But if design comes first, then the design (and code) must be developed with *seams*—locations in the detailed design where you can "insert test code that probes the state of your running software" and/or "isolate code under test from its production environment so that you can exercise it in a controlled testing context" [Wir09].

Sometimes referred to as "test hooks," seams must be consciously designed at the component level. To accomplish this, a designer must give thought to the tests that will be conducted to exercise the component. As Wirfs-Brock states: "In short, you need to provide appropriate test affordances—factoring your design in a way that lets test code interrogate and control the running system."

12.4 The Design Model

The design model can be viewed in two different dimensions as illustrated in Figure 12.4. The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process. The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to the figure, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design

Figure 12.4 Dimensions of the design model



models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

KEY POINT

The design model has four major elements: data, architecture, components, and interface.

The elements of the design model use many of the same UML diagrams[7] that were used in the analysis model. The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.

Quote:

"Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all."

Douglas Martin

You can apply design patterns (Chapter 16) at any point during design. These patterns enable you to apply design knowledge to domain-specific problems that have been encountered and solved by others.

12.4.1 Data Design Elements

Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

KEY POINT

At the architectural (application) level, data design focuses on files or databases; at the component level, data design considers the data structures that are required to implement local data objects.

The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role. Data design is discussed in more detail in Chapter 13.

12.4.2 Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into

7 Appendix 1 provides a tutorial on basic UML concepts and notation.

Quote:

"You can use an eraser on the drafting table or a sledge hammer on the construction site."

Frank Lloyd Wright

and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model [Sha96] is derived from three sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles (Chapter 13) and patterns (Chapter 16).

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces). Techniques for deriving specific elements of the architectural model are presented in Chapter 13.

12.4.3 Interface Design Elements

Quote:

"The public is more familiar with bad design than good design. It is, in effect, conditioned to prefer bad design, because that is what it lives with. The new becomes threatening, the old reassuring."

Paul Rand

The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.

There are three important elements of interface design: (1) the user interface (UI), (2) external interfaces to other systems, devices, networks, or other producers or consumers of information, and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

KEY POINT

There are three parts to the interface design element: the user interface, interfaces to system external to the application, and interfaces to components within the application.

UI design (increasingly called *usability design*) is a major software engineering action and is considered in detail in Chapter 15. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering (Chapter 8) and verified once the interface design commences.[8] The design of external interfaces should incorporate error checking and appropriate security features.

8 Interface characteristics can change with time. Therefore, a designer should ensure that the specification for the interface is accurate and complete.

Quote:

"Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen."

Leonardo DaVinci

WebRef

Extremely valuable information on UI design can be found at **www.useit.com**.

The design of internal interfaces is closely aligned with component-level design (Chapter 14). Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested.

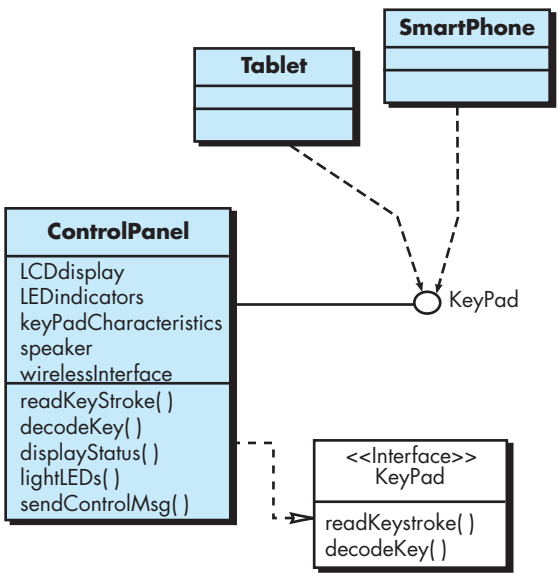
In some cases, an interface is modeled in much the same way as a class. In UML, an interface is defined in the following manner [OMG03a]: "An interface is a specifier for the externally-visible [public] operations of a class, component, or other classifier (including subsystems) without specification of internal structure." Stated more simply, an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.

For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a mobile platform (e.g., smartphone or tablet).

The **ControlPanel** class (Figure 12.5) provides the behavior associated with a keypad, and therefore, it must implement the operations *readKeyStroke ()* and *decodeKey ()*. If these operations are to be provided to other classes (in this case, **Tablet** and **SmartPhone**), it is useful to define an interface as shown in the figure. The interface, named **KeyPad**, is shown as an <<interface>> stereotype or as a small, labeled circle connected to the class with a line. The interface is defined with no attributes and the set of operations that are necessary to achieve the behavior of a keypad.

FIGURE 12.5

Interface representation for ControlPanel



Quote:

"A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools."

Douglas Adams

The dashed line with an open triangle at its end (Figure 12.5) indicates that the **ControlPanel** class provides **KeyPad** operations as part of its behavior. In UML, this is characterized as a *realization*. That is, part of the behavior of **ControlPanel** will be implemented by realizing **KeyPad** operations. These operations will be provided to other classes that access the interface.

12.4.4 Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Quote:

"The details are not the details. They make the design."

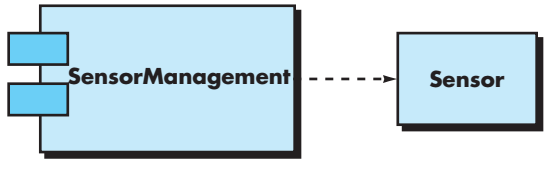
Charles Eames

Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 12.6. In this figure, a component named **SensorManagement** (part of the *SafeHome* security function) is represented. A dashed arrow connects the component to a class named **Sensor** that is assigned to it. The **SensorManagement** component performs all functions associated with *SafeHome* sensors including monitoring and configuring them. Further discussion of component diagrams is presented in Chapter 14.

The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode (a programming languagelike representation described in Chapter 14) or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

FIGURE 12.6

A UML component diagram



12.4.5 Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a home-based PC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system). In addition, limited functionality may be provided with mobile platforms.

KEY POINT

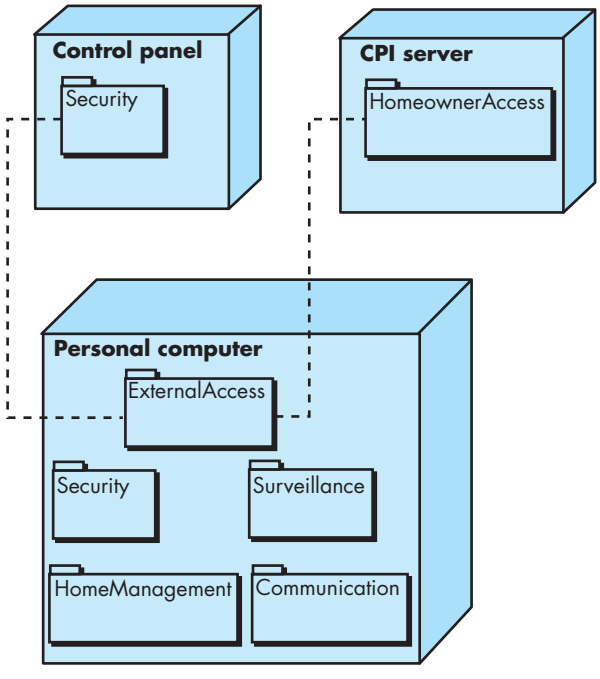
Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

During design, a UML deployment diagram is developed and then refined as shown in Figure 12.7. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.

The diagram shown in Figure 12.7 is in *descriptor form*. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further

FIGURE 12.7

A UML deployment diagram



identified. It could be a Mac, a Windows-based PC, a Linux-box or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific, named hardware configuration) is identified.

12.5 Summary

Software design commences as the first iteration of requirements engineering comes to a conclusion. The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product. The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it. Software designers must sift through many design alternatives and converge on a solution that best suits the needs of project stakeholders.

The design process moves from a "big picture" view of software to a more narrow view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Design concepts have evolved over the first 60 years of software engineering work. They describe attributes of computer software that should be present regardless of the software engineering process that is chosen, the design methods that are applied, or the programming languages that are used. In essence, design concepts emphasize the need for abstraction as a mechanism for creating reusable software components; the importance of architecture as a way to better understand the overall structure of a system; the benefits of pattern-based engineering as a technique for designing software with proven capabilities; the value of separation of concerns and effective modularity as a way to make software more understandable, more testable, and more maintainable; the consequences of information hiding as a mechanism for reducing the propagation of side effects when errors do occur; the impact of functional independence as a criterion for building effective modules; the use of refinement as a design mechanism; a consideration of aspects that crosscut system requirements; the application of refactoring for optimizing the design that is derived; the importance of object-oriented classes and the characteristics that are related to them; the need to use abstraction to reduce coupling between components, and the importance of design for testing.

The design model encompasses four different elements. As each of these elements is developed, a more complete view of the design evolves. The architectural element uses information derived from the application domain, the requirements model, and available catalogs for patterns and styles to derive a

complete structural representation of the software, its subsystems, and components. Interface design elements model external and internal interfaces and the user interface. Component-level elements define each of the modules (components) that populate the architecture. Finally, deployment-level design elements allocate the architecture, its components, and the interfaces to the physical configuration that will house the software.

Problems and Points to Ponder

12.1. Do you design software when you "write" a program? What makes software design different from coding?

12.2. If a software design is not a program (and it isn't), then what is it?

12.3. How do we assess the quality of a software design?

12.4. Examine the task set presented for design. Where is quality assessed within the task set? How is this accomplished? How are the quality attributes discussed in Section 12.2.1 achieved?

12.5. Provide examples of three data abstractions and the procedural abstractions that can be used to manipulate them.

12.6. Describe software architecture in your own words.

12.7. Suggest a design pattern that you encounter in a category of everyday things (e.g., consumer electronics, automobiles, appliances). Briefly describe the pattern.

12.8. Describe separation of concerns in your own words. Is there a case when a "divide and conquer" strategy may not be appropriate? How might such a case affect the argument for modularity?

12.9. When should a modular design be implemented as monolithic software? How can this be accomplished? Is performance the only justification for implementation of monolithic software?

12.10. Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.

12.11. How are the concepts of coupling and software portability related? Provide examples to support your discussion.

12.12. Apply a "stepwise refinement approach" to develop three different levels of procedural abstractions for one or more of the following programs: (1) Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check. (2) Iteratively solve for the roots of a transcendental equation. (3) Develop a simple task-scheduling algorithm for an operating system.

12.13. Consider the software required to implement a full navigation capability (using GPS) in a mobile, handheld communication device. Describe two or three crosscutting concerns that would be present. Discuss how you would represent one of these concerns as an aspect.

12.14. Does "refactoring" mean that you modify the entire design iteratively? If not, what does it mean?

12.15. Discuss what the dependency inversion principle is in your own words.

12.16. Why is design for testing so important?

12.17. Briefly describe each of the four elements of the design model.

Further Readings and Information Sources

Donald Norman has written three books (*Emotional Design: We Love (or Hate) Everyday Things,* Basic Books, 2005), (*The Design of Everyday Things,* Doubleday, 1990), and (*The Psychology of Everyday Things,* HarperCollins, 1988) that have become classics in the design literature and "must" reading for anyone who designs anything that humans use. Adams (*Conceptual Blockbusting,* 4th ed., Addison-Wesley, 2001) has written a book that is essential reading for designers who want to broaden their way of thinking. Finally, a classic text by Polya (*How to Solve It,* 2nd ed., Princeton University Press, 1988) provides a generic problem-solving process that can help software designers when they are faced with complex problems.

Books by Hanington and Martin (*Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions, Rockport,* 2012) and Hanington and Martin (*Universal Principles of Design: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design,* 2nd ed., Rockport, 2010) discuss design principles in general.

Following in the same tradition, Winograd et al. (*Bringing Design to Software,* Addison-Wesley, 1996) discusses software designs that work, those that don't, and why. A fascinating book edited by Wixon and Ramsey (*Field Methods Casebook for Software Design,* Wiley, 1996) suggests field research methods (much like those used by anthropologists) to understand how end users do the work they do and then design software that meets their needs. Holtzblatt (*Rapid Contextual Design: A How-to Guide to Key Techniques for User-Center Design,* Morgan Kaufman, 2004) and Beyer and Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs,* Academic Press, 1997) offer another view of software design that integrates the customer/user into every aspect of the software design process. Bain (*Emergent Design,* Addison-Wesley, 2008) couples patterns, refactoring, and test-driven development into an effective design approach.

Comprehensive treatment of design in the context of software engineering is presented by Otero (*Software Engineering Design: Theory and Practice,* Auerbach, 2012), Venit and Drake (*Prelude to Programming: Concepts and Design,* 5th ed., Addison-Wesley, 2010), Fox (*Introduction to Software Engineering Design,* Addison-Wesley, 2006), and Zhu (*Software Design Methodology,* Butterworth-Heinemann, 2005). McConnell (*Code Complete,* 2nd ed., Microsoft Press, 2004) presents an excellent discussion of the practical aspects of designing high-quality computer software. Robertson (*Simple Program Design,* 5th ed., Course Technology, 2006) presents an introductory discussion of software design that is useful for those beginning their study of the subject. Budgen (*Software Design,* 2nd ed., Addison-Wesley, 2004) introduces a variety of popular design methods, comparing and contrasting each. Fowler and his colleagues (*Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999) discuss techniques for the incremental optimization of software designs. Rosenberg and Stevens (*Use Case Driven Object Modeling with UML,* Apress, 2007) discuss the development of object-oriented designs using use cases as a foundation.

A worthwhile historical survey of software design is contained in an anthology edited by Freeman and Wasserman (*Software Design Techniques,* 4th ed., IEEE, 1983). This tutorial reprints many of the classic papers that have formed the basis for current trends in software design. Measures of design quality, presented from both the technical and management perspectives, are considered by Card and Glass (*Measuring Software Design Quality,* Prentice Hall, 1990).

A wide variety of information sources on software design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software design and design engineering can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

13 Architectural Design

Key Concepts

agility and architecture 280
archetypes 269
architectural decisions 266
architectural description language 276
architectural descriptions 255
architectural design 267

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. This description is extended by Freeman [Fre80]:

> [D]esign is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.

Quick Look

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

architectural genres 257
architectural patterns 263
architectural styles 258
architecture 253
architecture conformance checking 279
refining the architecture 270

As we noted in Chapter 12, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

Methods required to create "coherent, well-planned representations" of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

13.1 Software Architecture

In their landmark book on the subject, Shaw and Garlan [Sha96] discuss software architecture in the following manner:

> Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

13.1.1 What Is Architecture?

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art.

Quote:

"The architecture of a system is a comprehensive framework that describes its form and structure—its components and how they fit together."

Jerrold Grochow

Architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

KEY POINT

Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

Quote:

"Marry your architecture in haste, repent at your leisure."

Barry Boehm

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what we call "architectural design") and the actions that are applied to derive the software design. As one reviewer of a past edition noted:

> There is a distinct difference between the terms *architecture* and *design*. A *design* is an instance of an *architecture* similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

Although we agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design. Design begins with a consideration of architecture.

WebRef

Useful pointers to many software architecture sites can be obtained at **http://www.ewita.com/links/softwareArchitectureLinks.htm.**

13.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.

KEY POINT

The architectural model provides a Gestalt view of the system, allowing the software engineer to examine it as a whole.

- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [Bas03].

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 13.2 through 13.6) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

13.1.3 Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Smolander, Rossi, and Purao [Smo08] have identified multiple metaphors, representing different views of the same architecture, that stakeholders use to understand the term *software architecture*. The *blueprint metaphor* seems to be most familiar to the stakeholders who write programs to implement a system. Developers regard architecture descriptions as a means of transferring explicit information from architects to designers to software engineers charged with producing the system components. The *language metaphor* views architecture as a facilitator of communication across stakeholder groups. This view is preferred by stakeholders with a high customer focus (e.g., managers or marketing experts). The architectural description needs to be concise and easy to understand since it forms the basis for negotiation particularly in determining system boundaries.

The *decision metaphor* represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance. Each of these properties can have a significant impact on the system design. Stakeholders (e.g., project managers) view architectural decisions as the basis for allocating project resources and work tasks. These decisions may affect the sequence of tasks and the structure of the software team. The *literature metaphor* is used to document architectural solutions constructed in the past. This view supports the construction of artifacts and the transfer of knowledge between designers and software maintenance staff. It also supports stakeholders whose concern is reuse of components and designs.

An architectural description of a software-based system must exhibit characteristics that combine these metaphors. Tyree and Akerman [Tyr05] note this when they write:

> Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding of the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects.

Each of these "wants" is reflected in a different metaphor represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems,* [IEE00], with the following objectives: (1) to establish a conceptual framework and vocabulary for use during the design of software architecture, (2) to provide detailed guidelines for representing an architectural description, and (3) to encourage sound architectural design practices. An *architectural description* (AD) represents multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

13.1.4 Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish a historical record that can be useful when design modifications must be made.

Grady Booch [Boo11a] writes that when setting out to build an innovative product, software engineers often feel compelled to plunge right in, build stuff, fix what doesn't work, improve what does work, and then repeat the process. After doing this a few times, they begin to recognize that an architecture should be defined and decisions associated with architectural choices must be stated explicitly. It may not be possible to predict the right choices before building a new product. However, if innovators find that architectural decisions are worth repeating after testing their prototypes in the field, then a *dominant design*[1] for

1 *Dominant design* describes an innovative software architecture or process that becomes an industry standard after a period of successful adaptation and use in the marketplace.

INFO

Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue: Describe the architectural design issues that are to be addressed.

Resolution: State the approach you've chosen to address the design issue.

Category: Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

Assumptions: Indicate any assumptions that helped shape the decision.

Constraints: Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

Alternatives: Briefly describe the architectural design alternatives that were considered and why they were rejected.

Argument: State why you chose the resolution over other alternatives.

Implications: Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

Related decisions: What other documented decisions are related to this decision?

Related concerns: What other requirements are related to this decision?

Work products: Indicate where this decision will be reflected in the architecture description.

Notes: Reference any team notes or other documentation that was used to make the decision.

this type of product may begin to emerge. Without documenting what worked and what did not, it is hard for software engineers to decide when to innovate and when to use previously created architecture.

13.2 ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, the architectural *genre* will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on. Within each general style, more specific styles might apply (Section 13.3). Each style would have a structure that can be described using a set of predictable patterns.

KEY POINT

A number of different architectural styles may be applicable to a specific genre (also called an application domain).

In his evolving *Handbook of Software Architecture* [Boo08], Grady Booch suggests the following architectural genres for software-based systems that include artificial intelligence, communications, devices, financial, games, industrial, legal, medical, military, operating systems, transportation, and utilities, among many others.

13.3 Architectural Styles

Quote:

"There is at the back of every artist's mind, a pattern or type of architecture."

G. K. Chesterton

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [Bas03].

? What is an architectural style?

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered (Chapter 36), the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components [Bos00].

WebRef

Attribute-based architectural styles (ABAS) can be used as building blocks for software architectures. Information can be obtained at **www.sei.cmu.edu/architecture/abas.html.**

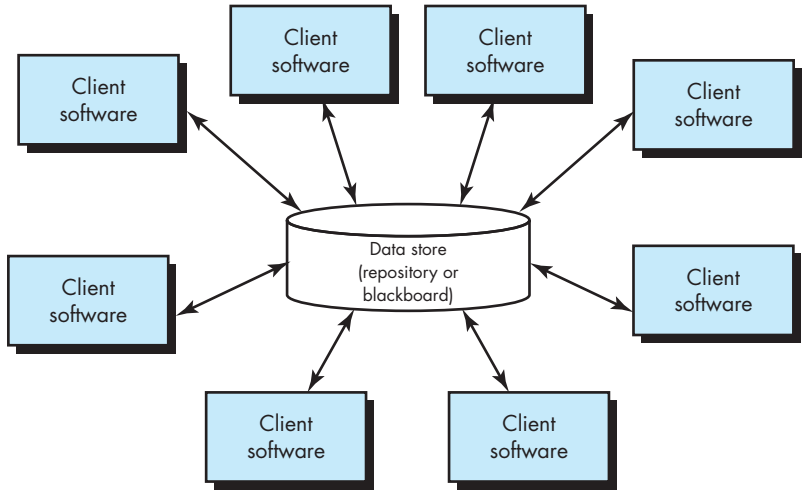
An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety, (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00], (3) architectural patterns (Section 13.3.2) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

13.3.1 A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

Data-Centered Architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components

FIGURE 13.1
Data-centered architecture



Quote:
"The use of patterns and styles of design is pervasive in engineering disciplines."
Mary Shaw and David Garlan

that update, add, delete, or otherwise modify data within the store. Figure 13.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

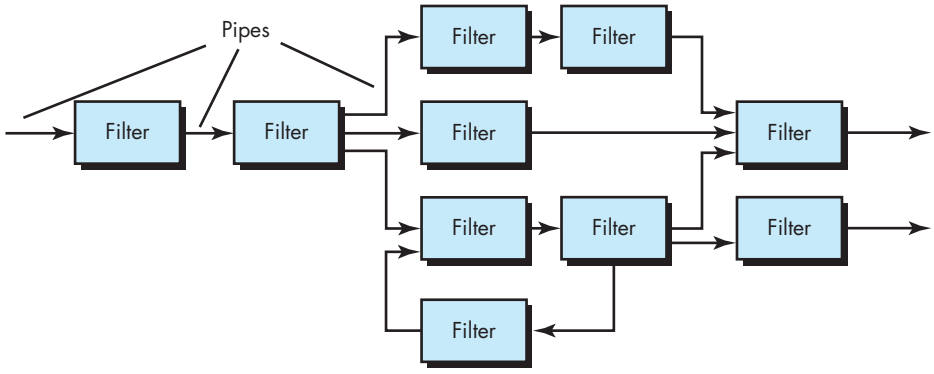
Data-centered architectures promote *integrability* [Bas03]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-Flow Architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 13.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed *batch sequential*. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

FIGURE 13.2

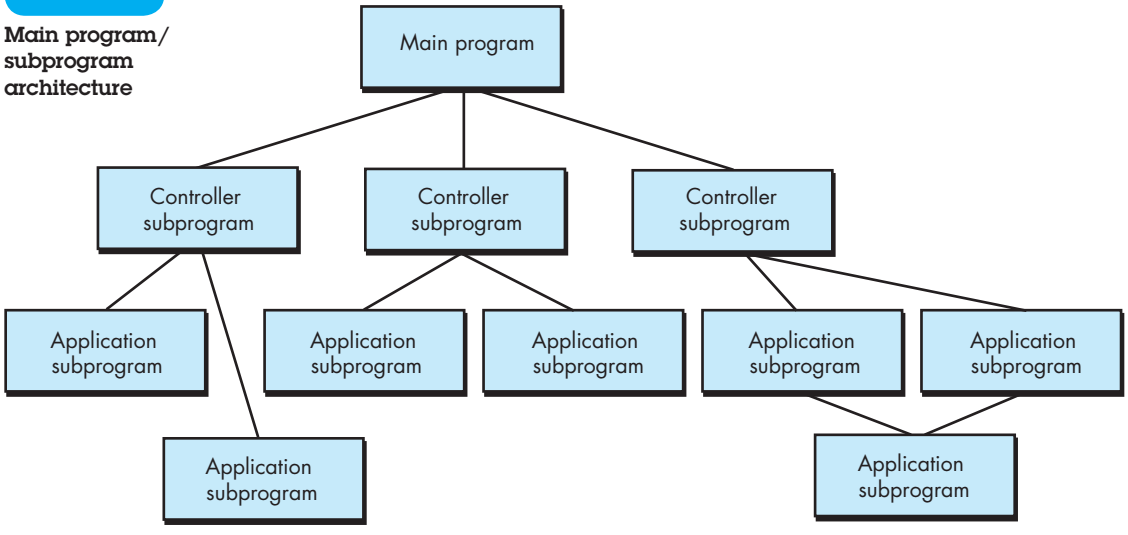
Data-flow architecture



Pipes and filters

FIGURE 13.3

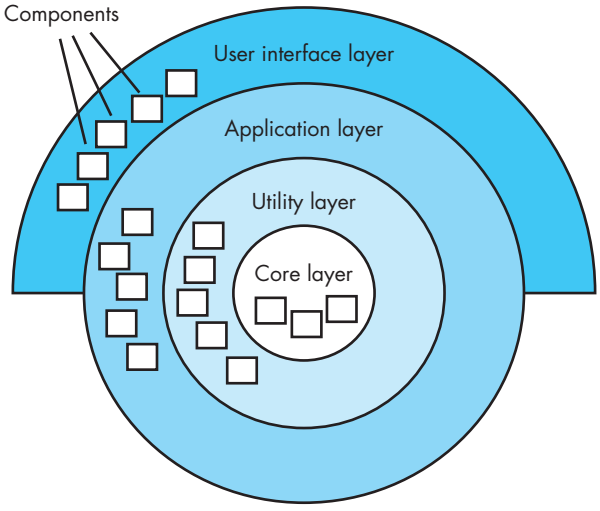
Main program/ subprogram architecture



Call and Return Architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub-styles [Bas03] exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.
- *Remote procedure call architectures.* The components of a main program/ subprogram architecture are distributed across multiple computers on a network.

FIGURE 13.4
Layered architecture



Object-Oriented Architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered Architectures. The basic structure of a layered architecture is illustrated in Figure 13.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available.[2] Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

Choosing the right architecture style can be tricky. Buschman [Bus10a] suggests two complementary concepts that can provide some guidance. *Problem frames* describe characteristics of recurring problems, without being distracted by references to details of domain knowledge or programming solution implementations. *Domain-driven design* suggests that the software design should

2 See [Roz11], [Tay09], [Bus07], [Gor06], or [Bas03], for a detailed discussion of architectural styles and patterns.

SafeHome

Choosing an Architectural Style

The scene: Jamie's cubicle, as design modeling begins.

The players: Jamie and Ed—members of the *SafeHome* software engineering team.

The conversation:

Ed (frowning): We've been modeling the security function using UML . . . you know classes, relationships, that sort of stuff. So I guess the object-oriented architecture[3] is the right way to go.

Jamie: But . . .?

Ed: But . . . I have trouble visualizing what an object-oriented architecture is. I get the call and return architecture, sort of a conventional process hierarchy, but OO . . . I don't know, it seems sort of amorphous.

Jamie (smiling): Amorphous, huh?

Ed: Yeah . . . what I mean is I can't visualize a real structure, just design classes floating in space.

Jamie: Well, that's not true. There are class hierarchies . . . think of the hierarchy (aggregation) we did for the **FloorPlan** object [Figure 12.3]. An OO architecture is a combination of that structure and the interconnections—you know, collaborations—between the classes. We can show it by fully describing the attributes and operations, the messaging that goes on, and the structure of the classes.

Ed: I'm going to spend an hour mapping out a call and return architecture; then I'll go back and consider an OO architecture.

Jamie: Doug'll have no problem with that. He said that we should consider architectural alternatives. By the way, there's absolutely no reason why both of these architectures couldn't be used in combination with one another.

Ed: Good. I'm on it.

reflect the domain and the domain logic of the business problem you want to solve with your application (Chapter 8).

A *problem frame* is a generalization of a class of problems that might be used to solve the problem at hand. There are five fundamental problem frames, and these are often associated with architectural styles: simple work pieces (tools), required behavior (data centered), commanded behavior (command processor), information display (observer), and transformation (pipe and filter variants).

Real-world problems often follow more than one problem frame, and as a consequence an architectural model may be a combination of different frames. For example, the model-view-controller (MVC) architecture used in WebApp design[4] might be viewed as combining two problem frames (command behavior and information display). In MVC the end user's command is sent from the browser window to a command processor (controller) which manages access to the content (model) and instructs the information rendering model (view) to translate it for display by the browser software.

3 It can be argued that the *SafeHome* architecture should be considered at a higher level than the architecture noted. *SafeHome* has a variety of subsystems—home monitoring functionality, the company's monitoring site, and the subsystem running in the owner's PC. Within subsystems, concurrent processes (e.g., those monitoring sensors) and event handling are prevalent. Some architectural decisions at this level are made during product engineering, but architectural design within software engineering may very well have to consider these issues.

4 The MVC architecture is considered in more detail in Chapter 17.

Domain modeling can influence the choice of architectural style, particularly the core properties of domain objects. The domain objects that represent physical objects (e.g., sensors or drives) should be treated differently from those representing logical objects (e.g., schedules or workflows). Physical objects must obey stringent constraints like connection limitations or use of consumable resources. Logical objects may have softer real-time behaviors that can be canceled or undone. Domain-driven design is often best supported by a layered architectural style. [Eva04]

13.3.2 Architectural Patterns

Quote:

"Maybe it's in the basement. Let me go upstairs and check."

M. C. Escher

As the requirements model is developed, you'll notice that the software must address a number of broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to many different customers and allow those customers to purchase our goods online?*

The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way you address the problem to be solved.

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Previously in this chapter, we noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns are presented in Chapter 16.

13.3.3 Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas03] provide insight into an architectural style:

? How do I assess an architectural style that has been derived?

> **Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

Evolutionary process models (Chapter 4) have become very popular. This implies the software architectures may need to evolve as each product increment is planned and implemented. In Chapter 12 we described this process as refactoring—improving the internal structure of the system without changing its external behavior.

13.4 Architectural Considerations

Buschmann and Henny [Bus10b, Bus10c] suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made.

? What issues should I consider as I develop a software architecture?

- **Economy**—Many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional requirements (e.g., reusability when it serves no purpose). The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility**—As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
- **Spacing—**Separation of concerns in a design without introducing hidden dependencies is a desirable design concept (Chapter 12) that is sometimes referred to as *spacing*. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility. Methods like domain-driven design can help to identify what to separate in a design and what to treat as a coherent unit.
- **Symmetry**—Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural

symmetry, consider a *customer account* object whose life cycle is modeled directly by a software architecture that requires both *open()* and *close()* methods. Architectural symmetry can be both structural and behavioral.

- **Emergence**—Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of the events that define the system's behavior is an emergent quality. It is very difficult to plan for every possible sequence of events. Instead the system architect should create a flexible system that accommodates this emergent behavior.

These considerations do not exist in isolation. They interact with each other and are moderated by each other. For example, spacing can be both reinforced and reduced by economy. Visibility can be balanced by spacing.

SafeHome

Evaluating Architectural Decisions

The scene: Jamie's cubicle, as design modeling continues.

The players: Jamie and Ed—members of the *SafeHome* software engineering team.

The conversation:

Ed: I finished my call-return architectural model of the security function.

Jamie: Great! Do you think it meets our needs?

Ed: It doesn't introduce any unneeded features, so it seems to be economic.

Jamie: How about visibility?

Ed: Well, I understand the model and there's no problem implementing the security requirements needed for this product.

Jamie: I get that you understand the architecture, but you may not be the programmer for this part of the project. I'm a little worried about spacing. This design may not be as modular as an object-oriented design.

Ed: Maybe, but that may limit our ability to reuse some of our code when we have to create the web-based version of this *SafeHome*.

Jamie: What about symmetry?

Ed: Well, that's harder for me to assess. It seems to me the only place for symmetry in the security function is adding and deleting PIN information.

Jamie: That will get more complicated when we add remote security features to the web-based product.

Ed: That's true, I guess.

[They both pause for a moment, pondering the architectural issues.]

Jamie: *SafeHome* is a real-time system, so state transition and sequencing of events will be tough to predict.

Ed: Yeah, but the emergent behavior of this system can be handled with a finite state model.

Jamie: How?

Ed: The model can be implemented based on the call-return architecture. Interrupts can be handled easily in many programming languages.

Jamie: Do you think we need to do the same kind of analysis for the object-oriented architecture we were initially considering?

Ed: I suppose it might be a good idea, since architecture is hard to change once implementation starts.

Jamie: It's also important for us to map the nonfunctional requirements besides security on top of these architectures to be sure they have been considered thoroughly.

Ed: Also, true.

The architectural description for a software product is not explicitly visible in the source code used to implement it. As a consequence, code modifications made over time (e.g., software maintenance activities) can cause slow erosion of the software architecture. The challenge for a designer is to find suitable abstractions for the architectural information. These abstractions have the potential to add structuring that improves readability and maintainability of the source code [Bro10b].

13.5 Architectural Decisions

Quote:

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines."

Frank Lloyd Wright

Decisions associated with system architecture capture key design issues and the rationale behind chosen architectural solutions. Some of these decisions include software system organization, selection of structural elements and their interfaces as defined by their intended collaborations, and the composition of these elements into increasingly larger subsystems [Kru09]. In addition, choices of architectural patterns, application technologies, middleware assets, and programming language can also be made. The outcome of the architectural decisions influences the system's nonfunctional characteristics and many of its quality attributes [Zim11] and can be documented with *developer notes*. These notes document key design decisions along with their justification, provide a reference for new project team members, and serve as a repository for lessons-learned.

In general, software architectural practice focuses on architectural views that represent and document the needs of various stakeholders. It is possible, however, to define a *decision view* that cuts across several views of information contained in traditional architectural representations. The decision view captures both the architecture design decisions and their rationale.

Service-oriented architecture decision (SOAD)[5] modeling [Zim11] is a knowledge management framework that provides support for capturing architectural decision dependencies in a manner that allows them to guide future development activities.

A *guidance model* contains knowledge about architectural decisions required when applying an architectural style in a particular application genre. It is based architectural information obtained from completed projects that employed the architectural style in that genre. The guidance model documents places where design problems exist and architectural decisions must be made, along with quality attributes that should be considered in selecting from among potential

5 SOAD is analogous to the use of architecture patterns discussed in Chapter 16. Further information can be obtained at: http://soadecisions.org/soad.htm

alternatives. Potential alternative solutions (with their pros and cons) from previous software applications are included to assist the architect in making the best decision possible.

The *decision model* documents both the architectural decisions required and records the decisions actually made on previous projects with their justifications. The guidance model feeds the architectural decision model in a *tailoring* step that allows the architect to delete irrelevant issues, enhance important issues, or add new issues. A decision model can make use of more than one guidance model and provides feedback to the guidance model after the project is completed. This feedback may be accomplished by *harvesting* lessons learned from project postmortem reviews.

13.6 Architectural Design

As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

? What is an archetype?

An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

A number of questions [Boo11b] must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? What visualizations might there be to help emphasize areas of risk? How can hidden system design patterns be made more obvious to other developers? Can multiple viewpoints show the best way to refactor specific parts of the system? Can design trade-offs be represented in a meaningful way? If a diagrammatic representation of software architecture answers these questions, it will have value to software engineers that use it.

13.6.1 Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 13.5.

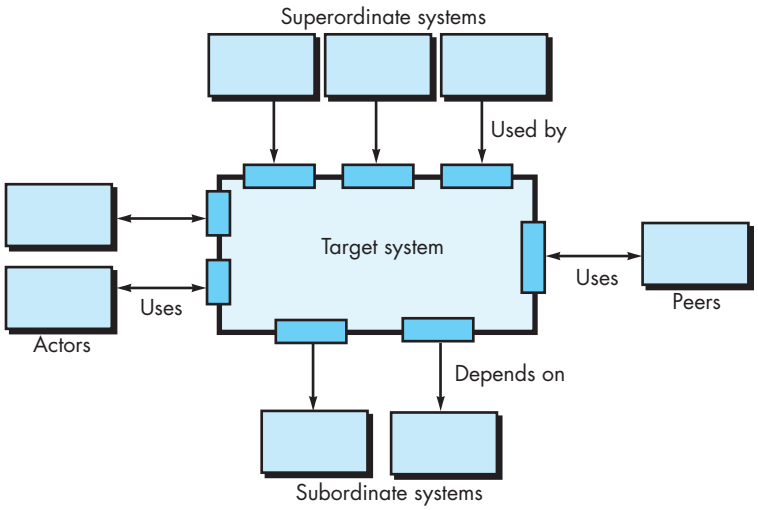


FIGURE 13.5
Architectural context diagram
Source: Adapted from [Bos00].

? How do systems interoperate with one another?

Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as:

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the *SafeHome* product. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 13.6. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that producer and consume information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

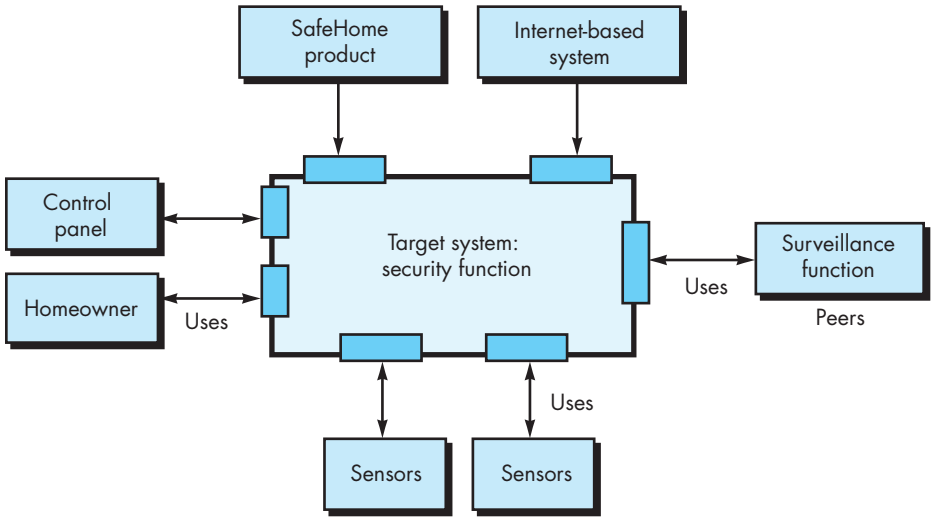


FIGURE 13.6 Architectural context diagram for the *SafeHome* security function

As part of the architectural design, the details of each interface shown in Figure 13.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

13.6.2 Defining Archetypes

KEY POINT
Archetypes are the abstract building blocks of an architectural design.

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

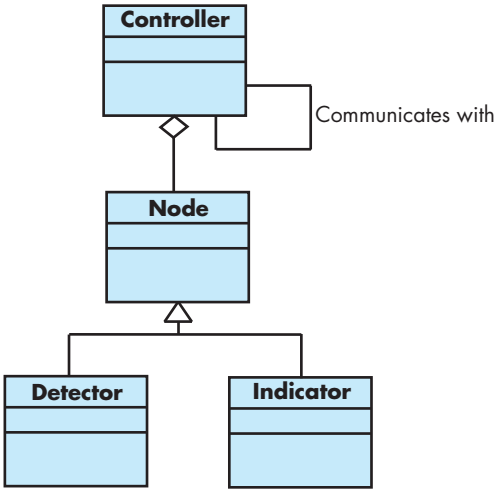


FIGURE 13.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

Each of these archetypes is depicted using UML notation as shown in Figure 13.7. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

13.6.3 Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model.[6] These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

The interfaces depicted in the architecture context diagram (Section 13.6.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

Quote:
"The structure of a software system provides the ecology in which code is born, matures, and dies. A well-designed habitat allows for the successful evolution of all the components needed in a software system."
R. Pattis

6 If a conventional (non-object-oriented) approach is chosen, components may be derived from the subprogram calling hierarchy (see Figure 13.3).

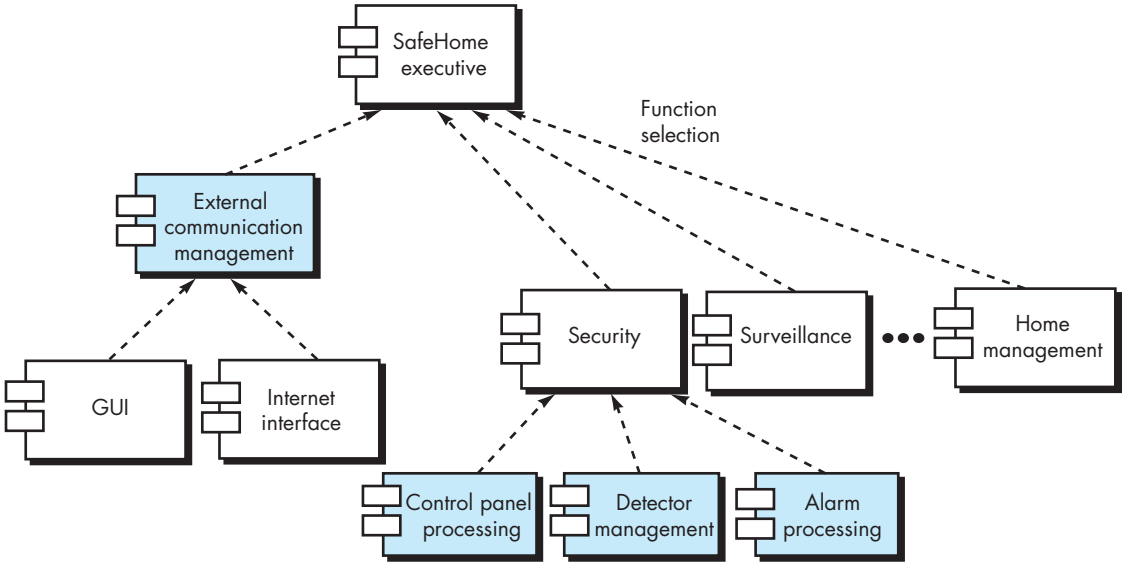
Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

- *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design (Chapter 14).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 13.8. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management*

FIGURE 13.8 **Overall architectural structure for *SafeHome* with top-level components**



component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

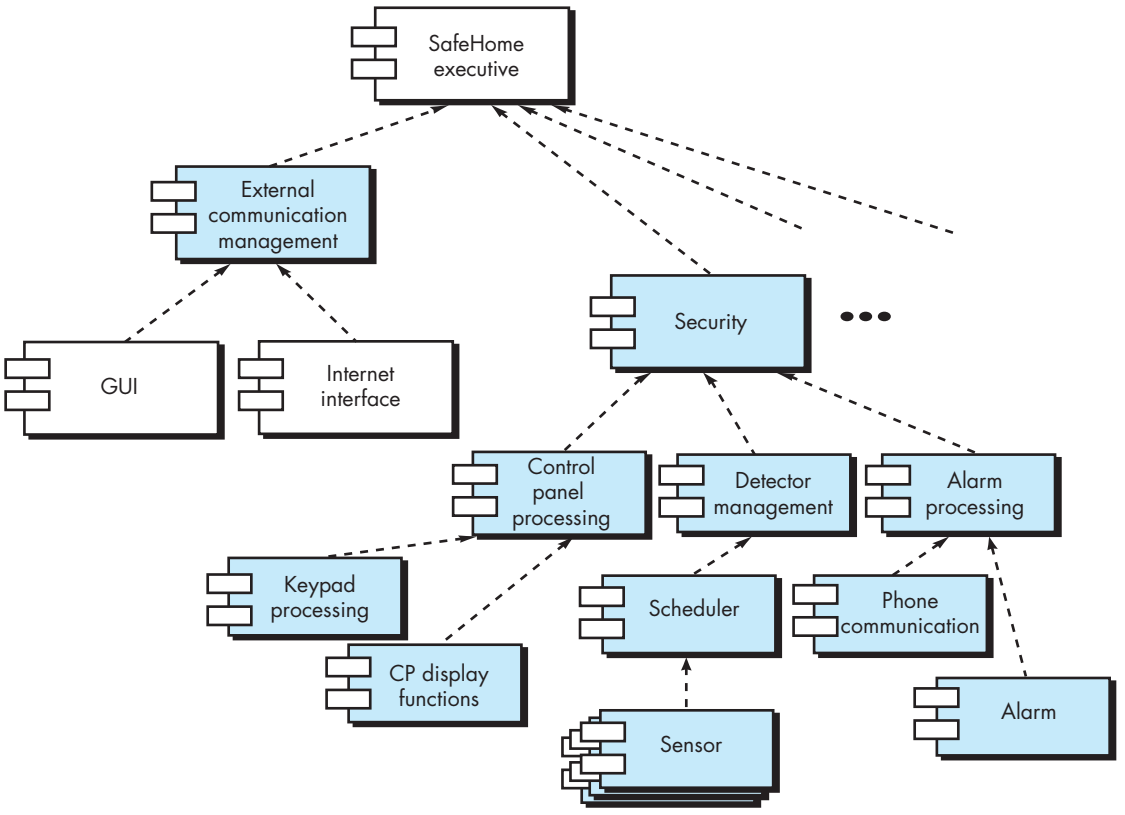
13.6.4 Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure 13.9 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 13.8 are elaborated to show additional detail. For example, the *detector management* component interacts with

FIGURE 13.9 An instantiation of the security function with component elaboration



a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 13.8.

SOFTWARE TOOLS

Architectural Design

Objective: Architectural design tools model the overall software structure by representing component interface, dependencies and relationships, and interactions.

Mechanics: Tool mechanics vary. In most cases, architectural design capability is part of the functionality provided by automated tools for analysis and design modeling.

Representative Tools:[7]

Adalon, developed by Synthis Corp. (**www.synthis.com**), is a specialized design tool for the design and construction of specific Web-based component architectures.

ObjectiF, developed by microTOOL GmbH (**www.microtool.de/objectiF/en/**), is a UML-based design tool that leads to architectures (e.g., Coldfusion, J2EE, Fusebox) amenable to component-based software engineering (Chapter 14).

Rational Rose, developed by Rational (**http://www-01.ibm.com/software/rational/**), is a UML-based design tool that supports all aspects of architectural design.

13.6.5 Architectural Design for Web Apps

WebApps[8] are client-server applications typically structured using multilayered architectures, including a user interface or view layer, a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and a content or model layer that may also contain the business rules for the WebApp.

The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine (usually a personal computer or mobile device). Data layers reside on a server. Business rules can be implemented using a server-based scripting language such as PHP or a client-based scripting language such as javascript. An architect will examine requirements for security and usability to determine which features should be allocated to the client or server.

The architectural design of a WebApp is also influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client. The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures. The physical location of media and other content resources also influences the architectural choices made by software engineers.

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8 WebApp design is discussed in more detail in Chapter 17.

13.6.6 Architectural Design for Mobile Apps

Mobile apps[9] are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. With mobile apps you have the choice of building a thin Web-based client or a rich client. With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server. With a rich client all three layers may reside on the mobile device itself.

Mobile devices differ from one another in terms of their physical characteristics (e.g., screen sizes, input devices), software (e.g., operating systems, language support), and hardware (e.g., memory, network connections). Each of these attributes shapes the direction of the architectural alternatives that can be selected. Meier and his colleagues [Mei09] suggest a number of considerations that can influence the architectural design of a mobile app: (1) the type of web client (thin or rich) to be built, (2) the categories of devices (e.g., smartphones, tablets) that are supported, (3) the degree of connectivity (occasional or persistent) required, (4) the bandwidth required, (5) the constraints imposed by the mobile platform, (6) the degree to which reuse and maintainability are important, and (7) device resource constraints (e.g., battery life, memory size, processor speed).

13.7 Assessing Alternative Architectural Designs

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state:

> To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn't it be nice to know in advance if you've placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you're buying a system or paying for its development, wouldn't you like to have some assurance that it's started off down the right path? If you're the architect yourself, wouldn't you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Indeed, answers to these questions would have value. Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, we present two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

9 Mobile app design is discussed in more detail in Chapter 18.

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [Kaz98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. *Collect scenarios.* A set of use cases (Chapters 8 and 9) is developed to represent the system from the user's point of view.
2. *Elicit requirements, constraints, and environment description.* This information is required as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. *Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.* The architectural style(s) should be described using one of the following architectural views:
   - *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.
   - *Process view* for analysis of system performance.
   - *Data flow view* for analysis of the degree to which the architecture meets functional requirements.
4. *Evaluate quality attributes by considering each attribute in isolation.* The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.* This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points*.
6. *Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.* The SEI describes this approach in the following manner [Kaz98]:

   > Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of

the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.[10]

SafeHome

Architecture Assessment

The scene: Doug Miller's office as architectural design modeling proceeds.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team and Doug Miller, manager of the software engineering group.

The conversation:

Doug: I know you guys are deriving a couple of different architectures for the *SafeHome* product, and that's a good thing. I guess my question is, how are we going to choose the one that's best?

Ed: I'm working on a call and return style and then either Jamie or I are going to derive an OO architecture.

Doug: Okay, and how do we choose?

Jamie: I took a CS course in design in my senior year, and I remember that there are a number of ways to do it.

Vinod: There are, but they're a bit academic. Look, I think we can do our assessment and choose the right one using use cases and scenarios.

Doug: Isn't that the same thing?

Vinod: Not when you're talking about architectural assessment. We already have a complete set of use cases. So we apply each to both architectures and see how the system reacts, how components and connectors work in the use case context.

Ed: That's a good idea. Make sure we didn't leave anything out.

Vinod: True, but it also tells us whether the architectural design is convoluted, whether the system has to twist itself into a pretzel to get the job done.

Jamie: Aren't scenarios just another name for use cases?

Vinod: No, in this case a scenario implies something different.

Doug: You're talking about a quality scenario or a change scenario, right?

Vinod: Yes. What we do is go back to the stakeholders and ask them how *SafeHome* is likely to change over the next, say, three years. You know, new versions, features, that sort of thing. We build a set of change scenarios. We also develop a set of quality scenarios that define the attributes we'd like to see in the software architecture.

Jamie: And we apply them to the alternatives.

Vinod: Exactly. The style that handles the use cases and scenarios best is the one we choose.

13.7.1 Architectural Description Languages

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [Hof01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language-based techniques for architectural design have been

10 The *software architecture analysis method* (SAAM) is an alternative to ATAM and is well worth examining by those readers interested in architectural analysis. A paper on SAAM can be downloaded from www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html.

SOFTWARE TOOLS

Architectural Description Languages

The following summary of a number of important ADLs was prepared by Rickard Land [Lan02] and is reprinted with the author's permission. It should be noted that the first five ADLs listed have been developed for research purposes and are not commercial products.

xArch (http://www.isr.uci.edu/projects/xarchuci/) a standard, extensible XML-based representation for software architectures.

UniCon (www.cs.cmu.edu/~UniCon) is "an architectural description language intended to aid designers in defining software architectures in terms of abstractions that they find useful."

Wright (www.cs.cmu.edu/~able/wright/) is a formal language including the following elements: *components* with *ports*, *connectors* with *roles*, and *glue* to attach roles to ports. Architectural styles can be formalized in the language with predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

Acme (www.cs.cmu.edu/~acme/) can be seen as a second-generation ADL, in that its intention is to identify a kind of least common denominator for ADLs.

UML (www.uml.org/) includes many of the artifacts needed for architectural descriptions—processes, nodes, views, etc. For informal descriptions, UML is well suited just because it is a widely understood standard. It, however, lacks the full strength needed for an adequate architectural description.

established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

13.7.2 Architectural Reviews

Architectural reviews are a type of specialized technical review (Chapter 20) that provide a means of assessing the ability of a software architecture to meet the system's quality requirements (e.g., scalability or performance) and to identify any potential risks. Architectural reviews have the potential to reduce project costs by detecting design problems early.

Unlike requirements reviews that involve representatives of all stakeholders, architecture reviews often involve only software engineering team members supplemented by independent experts. The most common architectural review techniques used in industry are: experienced-based reasoning,[11] prototype evaluation, scenario review (Chapter 9), and use of checklists.[12] Many architectural reviews occur early in the project life cycle, they should also occur after new components or packages are acquired in component-based design (Chapter 14). Software engineers who conduct architectural reviews note that architectural work products are sometimes missing or inadequate, thereby making reviews difficult to complete [Bab09].

11 *Experience-based reasoning* compares the new software architecture to an architecture used to create a similar system in the past.

12 Representative checklists can be found at http://www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm

13.8 Lessons Learned

WebRef

Examples of software architectural design lessons learned can be found at **http://www.sei.cmu.edu/library/abstracts/news-at-sei/01feature200707.cfm**

Software-based systems are built by people with a variety of different needs and points of view. Therefore, a software architect should build consensus among members of the software team (and other stakeholders) in order to achieve the architectural vision for the final software product [Wri11].

Architects often focus on the long-term impact of the system's nonfunctional requirements as the architecture is created. Senior managers assess the architecture within the context of business goals and objectives. Project managers are often driven by short-term considerations of delivery dates and budget. Software engineers are often focused on their own technology interests and feature delivery. Each of these (and other) constituencies should work to achieve consensus that the software architecture chosen has distinct advantages over any other alternatives.

WebRef

A discussion of pattern-based architecture reviews appears at **http://www.infoq.com/articles/ieee-pattern-based-architecture-reviews**

Wright [Wri11] suggests the use of several *decision analysis and resolution* (DAR) methods that may help to counteract some hindrances to collaboration. These methods can help increase active team member participation and increase the likelihood of their buy-in to the final decision. DAR methods help team members to consider several viable architectural alternatives in an objective manner. Three representative examples of DAR methods are:

- **Chain of causes.** This technique is a form of root cause[13] analysis in which the team defines an architectural goal or effect and then enunciates the related actions that will cause the goal to be achieved.
- **Ishikawa fishbone.**[14] This is a graphical technique that identifies the many possible actions or causes required to achieve a desired architectural goal.
- **Mind mapping or spider diagrams.**[15] This diagram is used to represent words, concepts, tasks, or software engineering artifacts arranged around a central key word, constraint, or requirement.

13.9 Pattern-based Architecture Review

Formal technical reviews (Chapter 20) can be applied to software architecture and provide a means for managing system quality attributes, uncovering errors, and avoiding unnecessary rework. However, in situations in which short build cycles, tight deadlines, volatile requirements, and/or small teams are the norm,

13 Further information can be obtained at: http://www.thinkreliability.com/Root-Cause-Analysis-CM-Basics.aspx

14 Further information can be obtained at: http://asq.org/learn-about-quality/cause-analysis-tools/overview/fishbone.html

15 Further information can be obtained at: http://mindmappingsoftwareblog.com/5-best-mind-mapping-programs-for-brainstorming/

a lightweight architectural review process known as *pattern-based architecture review* (PBAR) might be the best option.

PBAR is an evaluation method that leverages the relationship between architectural patterns[16] and software quality attributes. A PBAR is a face-to-face audit meeting involving all developers and other interested stakeholders. An external reviewer with expertise in architecture, architecture patterns, quality attributes, and the application domain is also in attendance. The system architect is the primary presenter.

A PBAR should be scheduled after the first working prototype or *walking skeleton*[17] is completed. The PBAR encompasses the following iterative steps [Har11]:

1. Identify and discuss the quality attributes most important to the system by walking through the relevant use cases (Chapter 9).
2. Discuss a diagram of the system's architecture in relation to its requirements.
3. Help the reviewers identify the architecture patterns used and match the system's structure to the patterns' structure.
4. Using existing documentation and past use cases, examine the architecture and quality attributes to determine each pattern's effect on the system's quality attributes.
5. Identify and discuss all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of the issues uncovered during the meeting and makes appropriate revisions to the walking skeleton.

PBARs are well-suited to small, agile teams and require a relatively small amount of extra project time and effort. With its short preparation and review time, PBAR can accommodate changing requirements and short build cycles, and at the same time, help improve the team's understanding of the system architecture.

13.10 Architecture Conformance Checking

As the software process moves through design and into construction, software engineers must work to ensure that an implemented and evolving system conforms to its planned architecture. Many things (e.g., conflicting requirements,

16 An *architectural pattern* is a generalized solution to an architectural design problem with a specific set of conditions or constraints. Patterns are discussed in detail in Chapter 16.

17 A walking skeleton contains a baseline architecture that supports the functional requirements with the highest priorities in the business case and the most challenging quality attributes.

WebRef

An overview of architecture conformance checking appears at **http://www.cin.ufpe.br/~fcf3/Arquitetura%20de%20Software/arquitetura/getPDF3.pdf**

technical difficulties, deadline pressures) cause deviations from a defined architecture. If architecture is not checked for conformance periodically, uncontrolled deviations can cause *architecture erosion* and affect the quality of the system [Pas10].

Static architecture-conformance analysis (SACA) assesses whether an implemented software system is consistent with its architectural model. The formalism (e.g., UML) used to model the system architecture presents the static organization of system components and how the components interact. Often the architectural model is used by a project manager to plan and allocate work tasks, as well as to assess implementation progress.

SOFTWARE TOOLS

Architectural-Conformance Tools

Lattix Dependency Manager (http://www.lattix.com/). This tool includes a simple language to declare design rules that the implementation must follow, detects violations in design rules, and visually represents them as a dependency-structure matrix.

Source Code Query Languages (http://www.semmle.com/). This tool can be used to automate software development tasks such defining and checking architectural constraints and makes use of a Prolog-like to define recursive queries on the inheritance hierarchy of object-oriented systems.

Reflexion Models (http://www.iese.fraunhofer.de/en/competencies/architecture/tools_architecture.html#contentPar_textblockwithpics). The SAVE tool can be used to allow software engineers to build a high-level model that captures the architecture of a system and then define the relations between this model and the source code. SAVE will then identify missing or erroneous relations between the model and the code.

13.11 AGILITY AND ARCHITECTURE

In the view of some proponents of agile development, architectural design is equated with "big design upfront." In their view, this leads to unnecessary documentation and the implementation of unnecessary features. However, most agile developers do agree [Fal10] that it is important to focus on software architecture when a system is complex (i.e., when a product has a large number of requirements, many stakeholders, or wide geographic distribution). For this reason, there is a need to integrate new architectural design practices into agile process models.

WebRef

A discussion of the role of architecture in agile software processes **http://msdn.microsoft.com/enus/architecture/ff476940.aspx**

In order to make early architectural decisions and avoid the rework required and/or the quality problems encountered required when the wrong architecture is chosen, agile developers should anticipate architectural elements[18] and structure based on an emerging collection of user stories (Chapter 5). By creating an

18 An excellent discussion of architectural agility can be found in [Bro10a].

architectural prototype (e.g., a walking skeleton) and developing explicit architectural work products to communicate to the necessary stakeholders, an agile team can satisfy the need for architectural design.

Agile development gives software architects repeated opportunities to work closely with the business and technical teams to guide the direction of a good architectural design. Madison [Mad10] suggests the use of a hybrid framework that contains elements of Scrum, XP, and sequential project management.[19] In this framework up-front planning sets the architectural direction, but moves quickly into storyboarding [Bro10b].

During storyboarding the architect contributes architectural user stories to the project and works with the product owner to prioritize the architectural stories with the business user stories as "sprints" (work units) are planned. The architect works with the team during the sprint to ensure that the evolving software continues to show high architectural quality. If quality is high, the team is left alone to continue development on its own. If not, the architect joins the team for the duration of the sprint. After the sprint is completed, the architect reviews the working prototype for quality before the team presents it to the stakeholders in a formal sprint review. Well-run agile projects require the iterative delivery of work products (including architectural documentation) with each sprint. Reviewing the work products and code as it emerges from each sprint is a useful form of architectural review.

Responsibility-driven architecture (RDA) is a process that focuses on architectural decision-making. It addresses when and how architectural decisions should be made and who on the project team makes them. This approach also emphasizes the role of architect as being a servant-leader rather than an autocratic decision maker and is consistent with the agile philosophy. The architect acts as facilitator and focuses on how the development team works with stakeholder concerns from outside the team (e.g., business, security, infrastructure).

Agile teams insist on the freedom to make changes as new requirements emerge. Architects want to make sure that the important parts of the architecture were carefully considered and that developers have consulted the appropriate stakeholders. Both concerns may be satisfied by making use of a practice called *progressive sign-off* in which the evolving product is documented and approved as each successive prototype is completed [Bla10].

Using a process that is compatible with the agile philosophy provides verifiable sign-off for regulators and auditors, without preventing agile teams from making decisions as needed. At the end of the project the team has a complete set of work products, and the architecture has been reviewed for quality as it evolves.

19 Scrum and XP are agile process models and are discussed in Chapter 5.

13.12 Summary

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

A number of different architectural styles and patterns are available to the software engineer and may be applied within a given architectural genre. Each style describes a system category that encompasses a set of components that perform a function required by a system; a set of connectors that enable communication, coordination, and cooperation among components; constraints that define how components can be integrated to form the system; and semantic models that enable a designer to understand the overall properties of a system.

In a general sense, architectural design is accomplished using four distinct steps. First, the system must be represented in context. That is, the designer should define the external entities that the software interacts with and the nature of the interaction. Once context has been specified, the designer should identify a set of top-level abstractions, called archetypes, that represent pivotal elements of the system's behavior or function. After abstractions have been defined, the design begins to move closer to the implementation domain. Components are identified and represented within the context of an architecture that supports them. Finally, specific instantiations of the architecture are developed to "prove" the design in a real-world context.

Architectural design can coexist with agile methods by applying a hybrid architectural design framework that makes use of existing techniques derived from popular agile methods. Once an architecture is developed, it can be assessed to ensure conformance with business goals, software requirements, and quality attributes.

Problems and Points to Ponder

13.1. Using the architecture of a house or building as a metaphor, draw comparisons with software architecture. How are the disciplines of classical architecture and the software architecture similar? How do they differ?

13.2. Present two or three examples of applications for each of the architectural styles noted in Section 13.3.1.

13.3. Some of the architectural styles noted in Section 13.3.1 are hierarchical in nature and others are not. Make a list of each type. How would the architectural styles that are not hierarchical be implemented?

13.4. The terms *architectural style, architectural pattern,* and *framework* (not discussed in this book) are often encountered in discussions of software architecture. Do some research and describe how each of these terms differs for its counterparts.

13.5. Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 13.3.3.

13.6. Research the ATAM (using [Kaz98]) and present a detailed discussion of the six steps presented in Section 13.7.1.

13.7. If you haven't done so, complete Problem 9.5. Use the design approach described in this chapter to develop a software architecture for the PHTRS.

13.8. Use the architectural decision template from Section 13.1.4 to document one of the architectural decisions for PHTRS architecture developed in Problem 13.7.

13.9. Select a mobile application you are familiar with, assess it using the architecture considerations (economy, visibility, spacing, symmetry, emergence) from Section 13.4.

13.10. List the strengths and weakness of the PHTRS architecture you created for Problem 13.7.

13.11. Create a dependency structure matrix[20] for the software PHTRS architecture created for Problem 13.7.

13.12. Pick an agile process model from Chapter 5 and identify the architectural design activities that are included.

Further Readings and Information Sources

The literature on software architecture has exploded over the past decade. Varma (*Software Architecture: A Case Based Approach,* Pearson, 2013) presents architecture in the context of a series of case studies. Books by Bass and his colleagues (*Software Architecture in Practice,* 3rd ed., Addison-Wesley, 2012), Gorton (*Essential Software Architecture,* 2nd ed., Springer, 2011), Rozanski and Woods (*Software Systems Architecture,* 2nd ed., Addison-Wesley, 2011), Eeles and Cripps (*The Process of Software Architecting,* Addison-Wesley, 2009), Taylor and his colleagues (*Software Architecture,* Wiley, 2009), Reekie and McAdam (*A Software Architecture Primer,* 2nd ed., Angophora Press, 2006), and Albin (*The Art of Software Architecture,* Wiley, 2003), present worthwhile treatments of an intellectually challenging topic area.

Buschman and his colleagues (*Pattern-Oriented Software Architecture,* Wiley, 2007) and Kuchana (*Software Architecture Design Patterns in Java,* Auerbach, 2004) discuss pattern-oriented aspects of architectural design. Knoernschilf (*Java Application Architecture: Modularity Patterns with Examples Using OSGi*, Prentice Hall, 2012), Rozanski and Woods (*Software Systems Architecture*, 2nd ed., Addison-Wesley, 2011), Henderikson (*12 Essential Skills for Software Architects,* Addison-Wesley, 2011), Clements and his colleagues (*Documenting Software Architecture: View and Beyond,* 2nd ed., Addison-Wesley, 2010), Microsoft (*Microsoft Application Guide*, Microsoft Press, 2nd ed., 2009), Fowler (*Patterns of Enterprise Application Architecture,* Addison-Wesley, 2003), Bosch [Bos00], and Hofmeister and his colleagues [Hof00] provide in-depth treatments of software architecture.

Hennesey and Patterson (*Computer Architecture,* 5th ed., Morgan-Kaufmann, 2011) take a distinctly quantitative view of software architectural design issues. Clements and his colleagues (*Evaluating Software Architectures,* Addison-Wesley, 2002) consider the issues associated with the assessment of architectural alternatives and the selection of the best architecture for a given problem domain.

---

20 Use Wikipedia as a starting point to obtain further information about the DSM at: http://en.wikipedia.org/wiki/Design_structure_matrix.

Implementation-specific books on architecture address architectural design within a specific development environment or technology. Erl (*SOA Design Patterns*, Prentice Hall, 2009) and Marks and Bell (*Service-Oriented Architecture,* Wiley, 2006) discuss a design approach that links business and computational resources with the requirements defined by customers. Bambilla et al. (*Model-Driven Software Engineering in Practice*, Morgan Claypool, 2012) and Stahl and his colleagues (*Model-Driven Software Development,* Wiley, 2006) discuss architecture within the context of domain-specific modeling approaches. Radaideh and Al-ameed (*Architecture of Reliable Web Applications Software,* IGI Global, 2007) consider architectures that are appropriate for WebApps. Esposito (*Architecting Mobile Solutions for the Enterprise*, Microsoft Press, 2012) discusses architecting mobile applications. Clements and Northrop (*Software Product Lines: Practices and Patterns,* Addison-Wesley, 2001) address the design of architectures that support software product lines. Shanley (*Protected Mode Software Architecture,* Addison-Wesley, 1996) provides architectural design guidance for anyone designing PC-based real-time operating systems, multitask operating systems, or device drivers.

Current software architecture research is documented yearly in the *Proceedings of the International Workshop on Software Architecture,* sponsored by the ACM and other computing organizations, and the *Proceedings of the International Conference on Software Engineering*.

A wide variety of information sources on architectural design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to architectural design can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

14

Component-Level Design

Key Concepts

cohesion 296
component 312
adaptation 310
classifying 286
composition . . . 310
qualification . . . 309
WebApp 305
component-based development 308
content design . . . 306
coupling 298
dependency inversion principle 293
design for reuse . . . 312
design guidelines . . 295
domain engineering 308
interface segregation principle 294
Liskov substitution principle 293
object-oriented view 286
open-closed principle 292
process-related . . . 291
traditional components 307
traditional view . . . 288

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. In a famous lecture, Edsgar Dijkstra, a major contributor to our understanding of software design, stated [Dij72]:

> Software seems to be different from many other products, where as a rule higher quality implies a higher price. Those who want really reliable software will discover that they must find a means of avoiding the majority of bugs to start with, and as a result, the programming process will become cheaper . . . effective programmers . . . should not waste their time debugging—they should not introduce bugs to start with.

Although these words were spoken many years ago, they remain true today. As you translate the design model into source code, you should follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

Quick Look

What is it? A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it? A software engineer performs component-level design.

Why is it important? You have to be able to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations (i.e., the data, architectural, and interface designs). It provides a means for

assessing whether data structures, interfaces, and algorithms will work.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic. Design notation encompasses UML diagrams and supplementary forms. Procedural design is specified using a set of structured programming constructs. It is often possible to acquire existing reusable software components rather than building new ones.

What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

14.1 What Is a Component?

Quote:

"The details are not the details. They make the design."

Charles Eames

A *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG03a] defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

As we discussed in Chapter 13, components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

The true meaning of the term *component* will differ depending on the point of view of the software engineer who uses it. In the sections that follow, we examine three important views of what a component is and how it is used as design modeling proceeds.

KEY POINT

From an object-oriented viewpoint, a component is a set of collaborating classes.

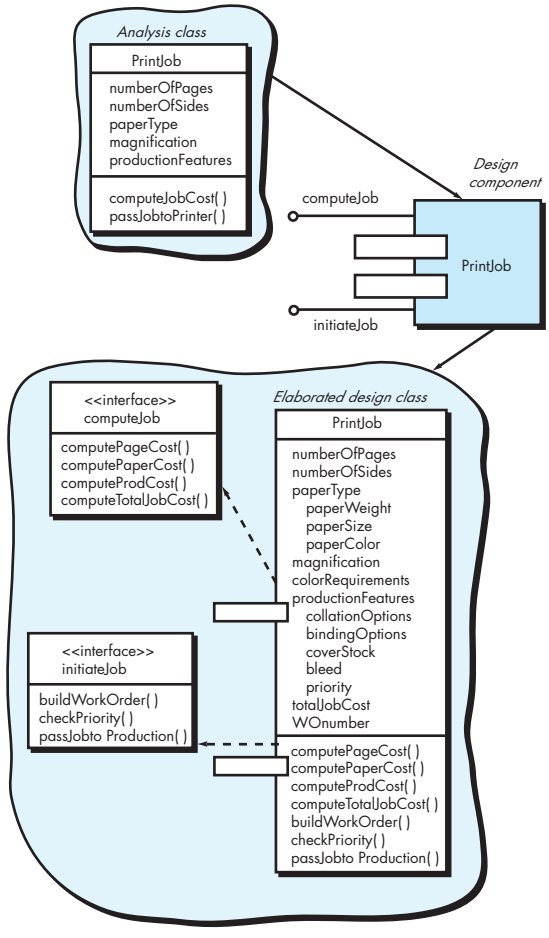
14.1.1 An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes.[1] Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To

1 In some cases, a component may contain a single class.

FIGURE 14.1

Elaboration of a design component



accomplish this, you begin with the analysis model and elaborate analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived. The attributes and operations defined during analysis are noted at the top of Figure 14.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation[2] shown in the middle right of

2 Readers who are unfamiliar with UML notation should refer to Appendix 1.

the figure. Note that **PrintJob** has two interfaces, *computeJob,* which provides job costing capability, and *initiateJob,* which passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

ADVICE

Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component).

Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob.** Referring to the lower right portion of Figure 14.1, the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

This elaboration activity is applied to every component defined as part of the architectural design. Once it is completed, further elaboration is applied to each attribute, operation, and interface. The data structures appropriate for each attribute must be specified. In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed. This procedural design activity is discussed later in this chapter. Finally, the mechanisms required to implement the interface are designed. For object-oriented software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

14.1.2 The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a *module,* resides within the software architecture and serves one of three important roles: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

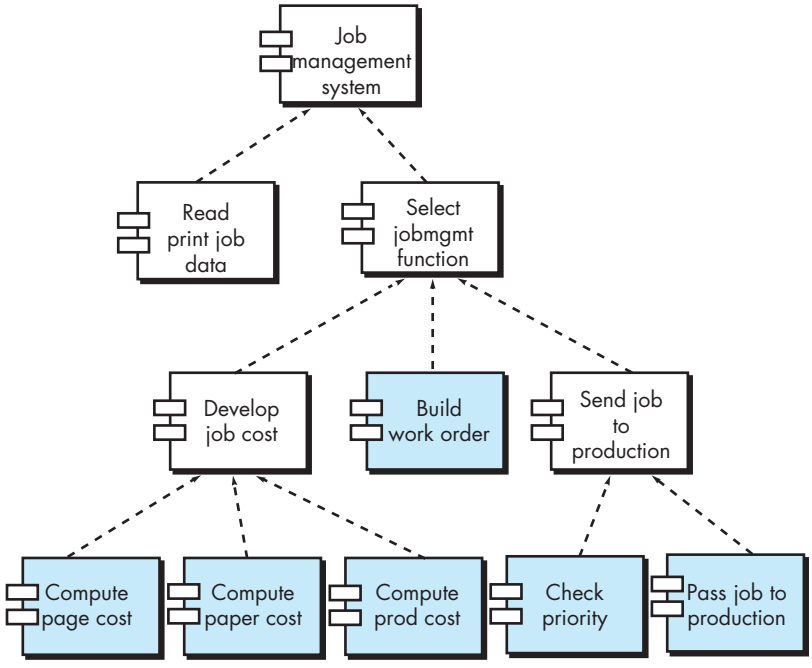
Quote:

"A complex system that works is invariably found to have evolved from a simple system that worked."

John Gall

Like object-oriented components, traditional software components are derived from the analysis model. In this case, however, the component elaboration element of the analysis model serves as the basis for the derivation. Each component represented the component hierarchy is mapped (Section 13.6) into a module hierarchy.

FIGURE 14.2 Structure chart for a traditional system



Control components (modules) reside near the top of the hierarchy (program architecture), and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence (Chapter 12) are applied as components are elaborated.

To illustrate this process of design elaboration for traditional components, again consider software to be built for a sophisticated print shop. A hierarchical architecture is derived and shown in Figure 14.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 14.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

During component-level design, each module in Figure 14.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 12. The behavior of the module is sometimes represented using a state diagram.

To illustrate this process, consider the module *ComputePageCost*. The intent of this module is to compute the printing cost per page based on specifications

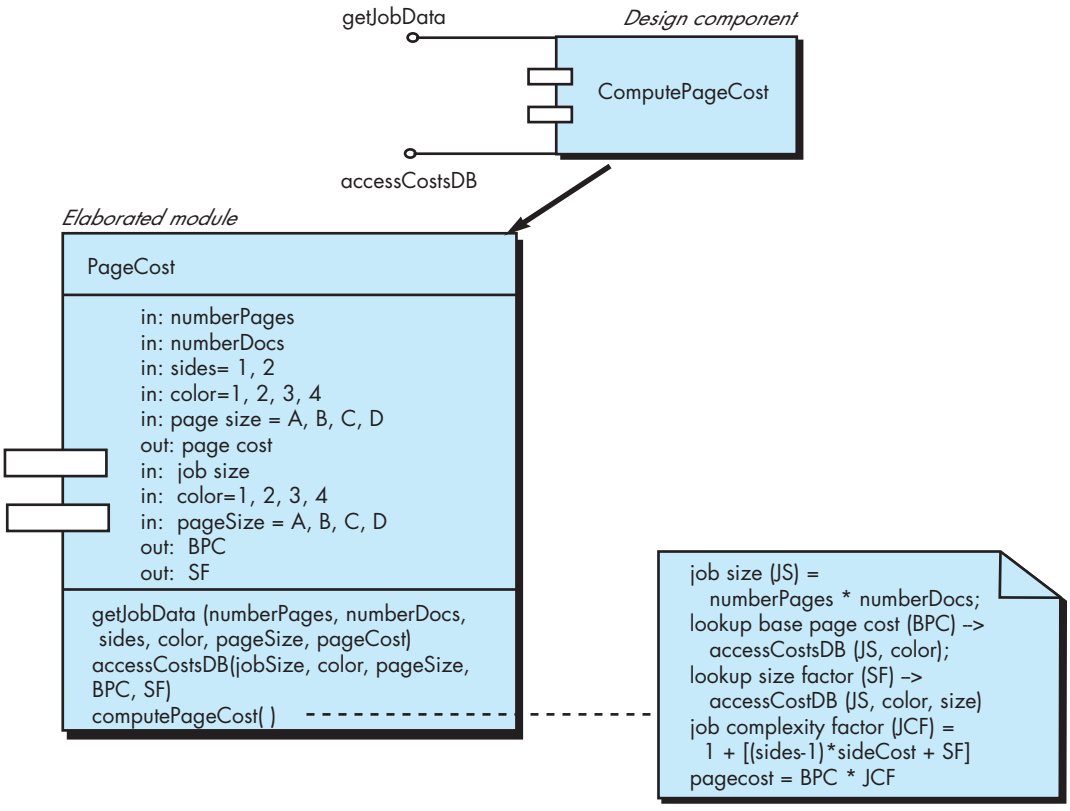
ADVICE

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

provided by the customer. Data required to perform this function are: **number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements,** and **size requirements.** These data are passed to *ComputePageCost* via the module's interface. *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

Figure 14.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the module *getJobData*, which allows all relevant data to be passed to the component, and a database interface, *accessCostsDB*, which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is elaborated to provide algorithm detail and interface detail (Figure 14.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as

FIGURE 14.3 **Component-level design for *ComputePageCost***



a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component.

14.1.3 A Process-Related View

The object-oriented and traditional views of component-level design presented in Sections 14.1.1 and 14.1.2 assume that the component is being designed from scratch. That is, you have to create a new component based on specifications derived from the requirements model. There is, of course, another approach.

Over the past three decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you. We discuss some of the important aspects of component-based software engineering (CBSE) later in Section 14.6.

INFO

Component-Based Standards and Frameworks

One of the key elements that lead to the success or failure of CBSE is the availability of component-based standards, sometimes called middleware. *Middleware* is a collection of infrastructure components that enable problem domain components to communicate with one another across a network or within a complex system. Software engineers who want to use component-based development as their software process can choose from among the following standards:

OMG CORBA— www.corba.org/

Microsoft COM— http://www.microsoft.com/com/default.mspx

Microsoft .NET— http://msdn.microsoft.com/en-us/netframework/default.aspx

Sun JavaBeans— http://www.oracle.com/technetwork/java/javaee/ejb/index.html

The websites noted present a wide array of tutorials, white papers, tools, and general resources on these important middleware standards.

14.2 Designing Class-Based Components

As we have already noted, component-level design draws on information developed as part of the requirements model (Chapters 9–11) and represented as part of the architectural model (Chapter 13). When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model. The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

14.2.1 Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur. You can use these principles as a guide as each software component is developed.

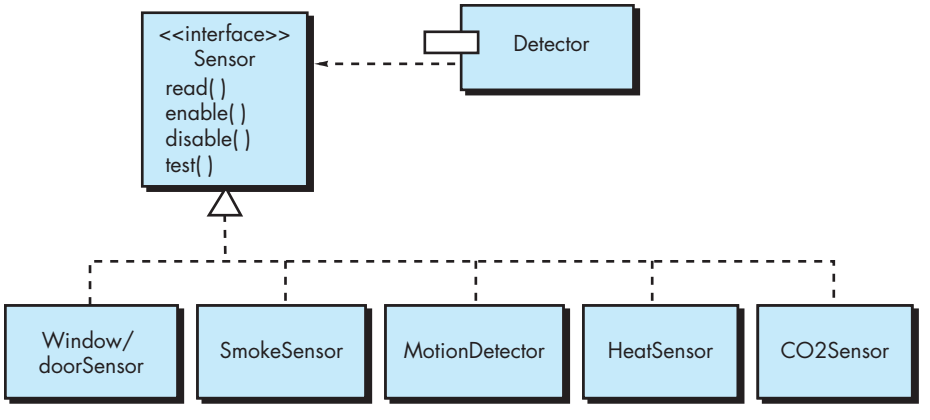
The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification"* [Mar00]. This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 14.4. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

FIGURE 14.4

Following the OCP



SafeHome

The OCP in Action

The scene: Vinod's cubicle.

The players: Vinod and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Vinod: I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.

Shakira (smirking): Not again, ¡eez!

Vinod: Yeah . . . and you're not going to believe what these guys have come up with.

Shakira: Amaze me.

Vinod (laughing): They call it a doggie angst sensor.

Shakira: Say what?

Vinod: It's for people who leave their pets home in apartments or condos or houses that are close to one another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.

Shakira: You're kidding me, right?

Vinod: Nope. Doug wants to know how much time it's going to take to add it to the security function.

Shakira (thinking a moment): Not much . . . look. [She shows Vinod Figure 14.4] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an appropriate component . . . uh, class, for it. No change to the **Detector** component at all.

Vinod: So I'll tell Doug it's no big deal.

Shakira: Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.

Vinod: That's not a bad thing, but you can implement now if he wants you to?

Shakira: Yeah, the way we designed the interface lets me do it with no hassle.

Vinod (thinking a moment): Have you ever heard of the open-closed principle?

Shakira (shrugging): Never heard of it.

Vinod (smiling): Not a problem.

The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes"* [Mar00]. This design principle, originally proposed by Barbara Liskov [Lis88], suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a *precondition* that must be true before the component uses a base class and a *postcondition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and postconditions.

ADVICE

If you dispense with design and hack out code, just remember that code is the ultimate "concretion." You're violating DIP.

Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions"* [Mar00]. As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface"* [Mar00]. There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

KEY POINT

Designing components for reuse requires more than good technical design. It also requires effective configuration control mechanisms (Chapter 29).

As an example, consider the **FloorPlan** class that is used for the *SafeHome* security and surveillance functions (Chapter 10). For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()* to place, show, group, and remove sensors from the floor plan. The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID()*. Hence, the ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them. The interface for security would encompass only the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()*. The interface for surveillance would incorporate the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice(),* along with *showFOV()* and *showDeviceID()*.

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design proceeds? Martin [Mar00] suggests additional packaging principles that are applicable to component-level design. These principles follow.

The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release"* [Mar00]. When classes or components are designed for reuse, an implicit contract is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP). *"Classes that change together belong together."* [Mar00] Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together"* [Mar00]. When one or more classes with a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operated without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

14.2.2 Component-Level Design Guidelines

In addition to the principles discussed in Section 14.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler [Amb02b] suggests the following guidelines:

What should we consider when we name components?

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class name **FloorPlan** is meaningful to everyone reading it regardless of technical background. On the other hand, infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning. If a linked list is to be managed as part of the **FloorPlan** implementation, the operation *manageList()* is appropriate, even if a nontechnical person might misinterpret it.[3]

You can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, <<**infrastructure**>> might be used to identify an infrastructure component, <<**database**>> could be used to identify a database that services one or more design classes or the entire system; <<**table**>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler [Amb02c] recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the

3 It is unlikely that someone from marketing or the customer organization (a nontechnical type) would examine detailed design information.

left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available. These recommendations are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, components' interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency. Following the philosophy of the OCP, this will help to make the system more maintainable.

14.2.3 Cohesion

In Chapter 12, we described cohesion as the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Lethbridge and Laganiére [Let01] define a number of different types of cohesion (listed in order of the level of the cohesion):[4]

Functional. Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns a result.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider, for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages as shown in Figure 14.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

ADVICE

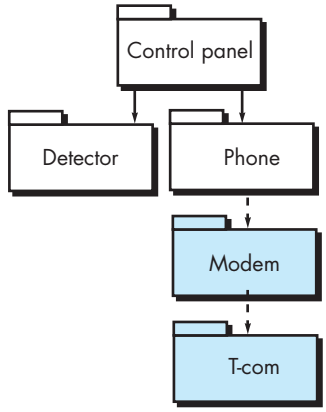
Although an understanding of the various levels of cohesion is instructive, it is more important to be aware of the general concept as you design components. Keep cohesion as high as is possible.

Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. You should strive to achieve these levels of cohesion whenever possible. It is important to note, however, that pragmatic design and implementation issues sometimes force you to opt for lower levels of cohesion.

4 In general, the higher the level of cohesion, the easier the component is to implement, test, and maintain.

FIGURE 14.5

Layer cohesion



SAFEHOME

Cohesion in Action

The scene: Jamie's cubicle.

The players: Jamie and Ed—members of the *SafeHome* software engineering team who are working on the surveillance function.

The conversation:

Ed: I have a first-cut design of the **camera** component.

Jamie: Wanna do a quick review?

Ed: I guess . . . but really, I'd like your input on something.

(Jamie gestures for him to continue.)

Ed: We originally defined five operations for **camera**. Look . . .

> *determineType()* tells me the type of camera.
>
> *translateLocation()* allows me to move the camera around the floor plan.
>
> *displayID()* gets the camera ID and displays it near the camera icon.
>
> *displayView()* shows me the field of view of the camera graphically.
>
> *displayZoom()* shows me the magnification of the camera graphically.

Ed: I've designed each separately, and they're pretty simple operations. So I thought it might be a good idea to combine all of the display operations into just one that's called *displayCamera()*—it'll show the ID, the view, and the zoom. Whaddaya think?

Jamie (grimacing): Not sure that's such a good idea.

Ed (frowning): Why? All of these little ops can cause headaches.

Jamie: The problem with combining them is we lose cohesion, you know, the *displayCamera()* op won't be single-minded.

Ed (mildly exasperated): So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.

Jamie: And what if marketing decides to change the way that we represent the view field?

Ed: I just jump into the *displayCamera()* op and make the mod.

Jamie: What about side effects?

Ed: Whaddaya mean?

Jamie: Well, say you make the change but inadvertently create a problem with the ID display.

Ed: I wouldn't be that sloppy.

Jamie: Maybe not, but what if some support person two years from now has to make the mod. He might not understand the op as well as you do, and, who knows, he might be sloppy.

Ed: So you're against it?

Jamie: You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.

Ed (thinking a moment): Maybe we'll go with separate display ops.

Jamie: Good decision.

ADVICE

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

14.2.4 Coupling

In earlier discussions of analysis and design, we noted that communication and collaboration are essential elements of any object-oriented system. There is, however, a darker side to this important (and necessary) characteristic. As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére [Let01] define a spectrum of coupling categories. For example, *content coupling* occurs when one component "surreptitiously modifies data that is internal to another component" [Let01]. This violates information hiding—a basic design concept. *Control coupling* occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result. *External coupling* occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

SafeHome

Coupling in Action

The scene: Shakira's cubicle.

The players: Vinod and Shakira—members of the *SafeHome* software team who are working on the security function.

The conversation:

Shakira: I had what I thought was a great idea . . . then I thought about it a little, and it seemed like a not so great idea. I finally rejected it, but I just thought I'd run it by you.

Vinod: Sure. What's the idea?

Shakira: Well, each of the sensors recognizes an alarm condition of some kind, right?

Vinod (smiling): That's why we call them sensors, Shakira.

Shakira (exasperated): Sarcasm, Vinod, you've got to work on your interpersonal skills.

Vinod: You were saying?

Shakira: Okay, anyway, I figured . . . why not create an operation within each sensor object called *makeCall()* that would collaborate directly with the **OutgoingCall** component, well, with an interface to the **OutgoingCall** component.

Vinod (pensive): You mean rather than having that collaboration occur out of a component like **ControlPanel** or something?

Shakira: Yeah . . . but then, I said to myself, that means that every sensor object will be connected to the **OutgoingCall** component, and that means that it's indirectly coupled to the outside world and . . . well, I just thought it made things complicated.

Vinod: I agree. In this case, it's a better idea to let the sensor interface pass info to the **ControlPanel** and let it initiate the outgoing call. Besides, different sensors might result in different phone numbers. You don't want the sensor to store that information because if it changes . . .

Shakira: It just didn't feel right.

Vinod: Design heuristics for coupling tell us it's not right.

Shakira: Whatever . . .

14.3 Conducting Component-Level Design

Quote:

"If I had more time, I would have written a shorter letter."

Blaise Pascal

Earlier in this chapter we noted that component-level design is elaborative in nature. You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated as described in Section 14.1.1.

ADVICE

If you're working in a non-OO environment, the first three steps focus on refinement of data objects and processing functions (transforms) identified as part of the analysis model.

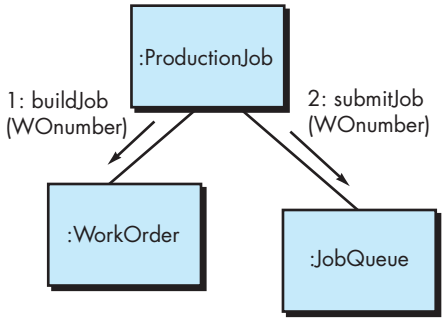
Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we have noted earlier, classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of

FIGURE 14.6

Collaboration diagram with messaging



messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

Figure 14.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During requirements modeling the messages are specified as shown in the figure. However, as design proceeds, each message is elaborated by expanding its syntax in the following manner [Ben02]:

[guard condition] sequence expression (return value) :=
message name (argument list)

where a **[guard condition]** is written in Object Constraint Language (OCL)[5] and specifies any set of conditions that must be met before the message can be sent; **sequence expression** is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; **(return value)** is the name of the information that is returned by the operation invoked by the message; **message name** identifies the operation that is to be invoked, and **(argument list)** is the list of attributes that are passed to the operation.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. . ." [Ben02]. Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. The elaboration of interfaces is illustrated in Figure 14.1. In essence, operations defined for the design class are categorized into one or more abstract classes. Every

5 OCL is discussed briefly in Appendix 1.

operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

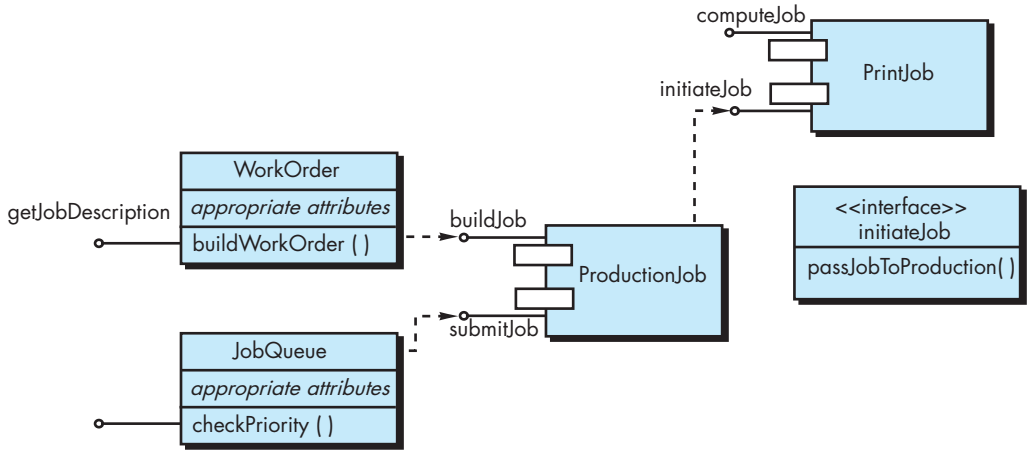
Referring to Figure 14.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. In actuality, it performs three different subfunctions—building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 14.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly single-minded.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

name : type-expression = initial-value {property string}

where **name** is the attribute name, **type expression** is the data type, **initial value** is the value that the attribute takes when an object is created, and **property-string** defines a property or characteristic of the attribute.

FIGURE 14.7 Refactoring interfaces and class definitions for PrintJob



During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 14.1, the attribute list for **PrintJob** lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defined using the UML attribute format noted. For example, **paperType-weight** is defined in the following manner:

paperType-weight: string = "A" { contains 1 of 4 values – A, B, C, or D}

which defines **paperType-weight** as a string variable initialized to the value A that can take on one of four values from the set {A, B, C, D}.

If an attribute appears repeatedly across a number of design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept (Chapter 12).

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. For example, the operation *computePaperCost()* noted in Figure 14.1 can be expanded in the following manner:

computePaperCost (weight, size, color): numeric

This indicates that *computePaperCost()* requires the attributes **weight**, **size**, and **color** as input and returns a value that is numeric (actually a dollar value) as output.

ADVICE

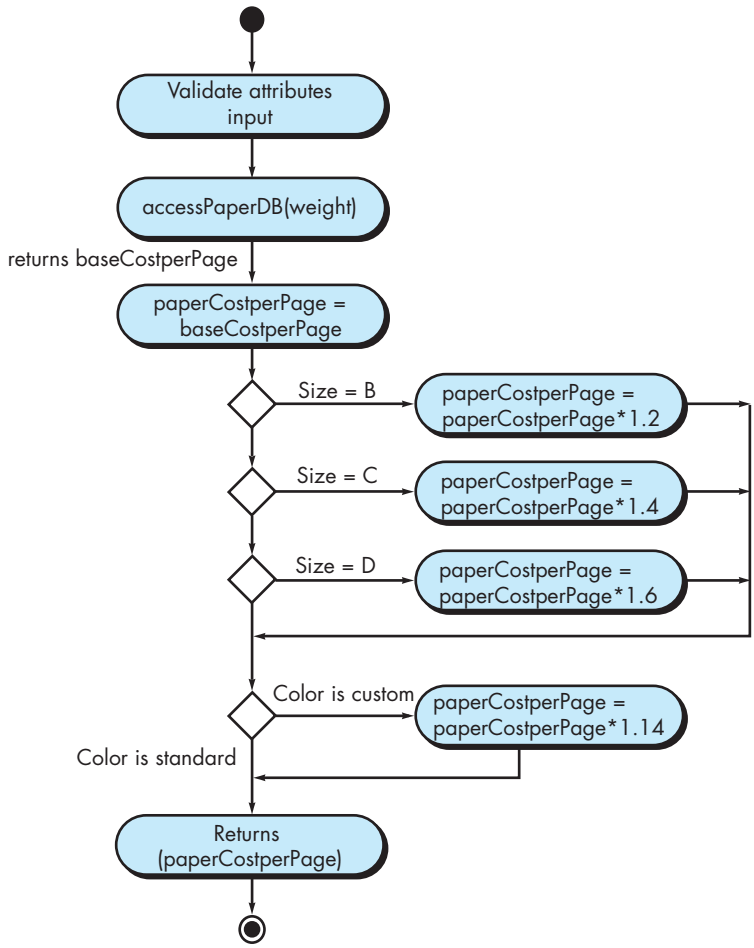
Use stepwise elaboration as you refine the component design. Always ask, "Is there a way this can be simplified and yet still accomplish the same result?"

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 14.8 depicts a UML activity diagram for *computePaperCost()*. When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code. An alternative approach—the use of pseudocode for design specification—is discussed in Section 14.5.3.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

FIGURE 14.8

UML activity diagram for *compute-PaperCost()*

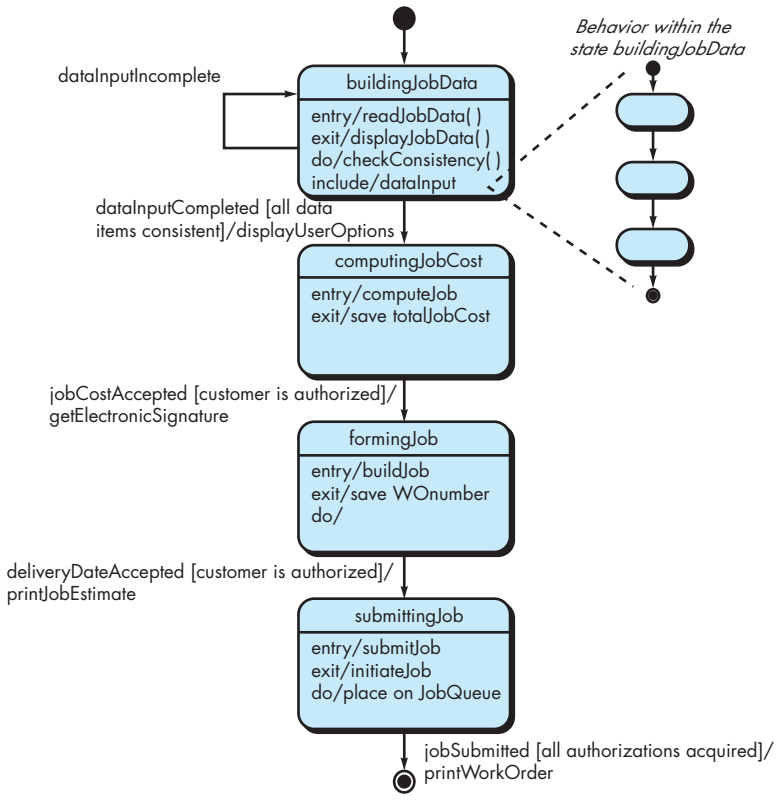


Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life. These use cases provide information that helps you to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) is represented using a UML statechart [Ben02] as illustrated in Figure 14.9.

FIGURE 14.9

Statechart fragment for PrintJob class



The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

Event-name (parameter-list) [guard-condition] / action expression

where **event-name** identifies the event, **parameter-list** incorporates data that are associated with the event, **guard-condition** is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and **action expression** defines an action that occurs as the transition takes place.

Referring to Figure 14.9, each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that occur while in the state, and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 14.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve) the print job cannot be submitted because there is no way to reach the *submittingJob* state.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams (Chapter 12) are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always consider alternatives. Throughout this book, we emphasize that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

In addition, you should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before settling on the final design model. Develop alternatives and consider each carefully, using the design principles and concepts presented in Chapter 12 and in this chapter.

14.4 Component-Level Design for WebApps

The boundary between content and function is often blurred when Web-based systems and applications (WebApps) are considered. Therefore, it is reasonable to ask: What is a WebApp component?

In the context of this chapter, a WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

14.4.1 Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelationships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation.[6] In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

14.4.2 Functional Design at the Component Level

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure consistency. In essence you begin by considering both the requirements model and the initial information architecture and then examining how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A *functional architecture* is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

14.5 Component-Level Design for Mobile Apps

In Chapter 13 we noted that mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. If you are building a mobile app as a thin Web-based client, the only components residing on a mobile device are those required to implement the user interface. Some mobile apps may incorporate the components required to implement the business and/or data layers on the mobile device subjecting these layers to the limitations of the physical characteristics of the device.

Considering the user interface layer first, it is important to recognize that a small display area requires the designer to be more selective in choosing the content (text and graphics) to be displayed. It may be helpful to tailor the content to a specific user group(s) and display only what each group needs. The business and data layers are often implemented by composing web or cloud service

6 Content components can also be reused in other WebApps.

components. If the components providing business and data services reside entirely on the mobile device, connectivity issues are not a significant concern. Intermittent (or missing) Internet connectivity must be considered when designing components that require access to current application data that reside on a networked server.

If a desktop application is being ported to a mobile device, the business-layer components may need to be reviewed to see if they meet nonfunctional requirements (e.g., security, performance, accessibility) required by the new platform. The target mobile device may lack the necessary processor speed, memory, or display real estate. The design of mobile applications is considered in greater detail in Chapter 18.

14.6 Designing Traditional Components

KEY POINT

Structured programming is a design technique that constrains logic flow to three constructs: sequence, condition, and repetition.

The foundations of component-level design for traditional software components[7] were formed in the early 1960s and were solidified with the work of Edsger Dijkstra ([Dij65], [Dij76b]) and others (e.g., [Boh66]. In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures. Complexity metrics (Chapter 30) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to

7 A traditional software component implements an element of processing that addresses a function or subfunction in the problem domain or some capability in the infrastructure domain. Often called modules, procedures, or subroutines, traditional components do not encapsulate data in the same way that object-oriented components do.

recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties.

14.7 Component-Based Development

In the software engineering context, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer-based systems must be built in very short time periods and demand a more organized approach to reuse.

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components." Considering this description, a number of questions arise. Is it possible to construct complex systems by assembling them from a catalog of reusable software components? Can this be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is management willing to incur the added expense associated with creating reusable software components? Can the library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it? Can existing components be found by those who need them? Increasingly, the answer to each of these questions is yes.

Quote:

"Domain engineering is about finding commonalities among systems to identify components that can be applied to many systems and to identify program families that are positioned to take fullest advantage of those components."

Paul Clements

14.7.1 Domain Engineering

The intent of *domain engineering* is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain.[8] The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

The overall approach to *domain analysis* is often characterized within the context of object-oriented software engineering. The steps in the process are: (1) define the domain to be investigated, (2) categorize the items extracted from the domain, (3) collect a representative sample of applications in the domain, (4) analyze each application in the sample and define analysis classes, and (5) develop a

8 In Chapter 13 we referred to architectural genres that identify specific application domains.

ADVICE

The analysis process we discuss in this section focuses on reusable components. However, the analysis of complete COTS systems (e.g., e-commerce Apps, sales force automation Apps) can also be a part of domain analysis.

requirements model for the classes. It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

14.7.2 Component Qualification, Adaptation, and Composition

Domain engineering provides the library of reusable components that are required for CBSE. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties.

Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

Component Qualification. Component qualification ensures that a candidate component will perform the function required, will properly "fit" into the architectural style (Chapter 13) specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

Design by contract is a technique that focuses on defining clear and verifiable component interface specifications, thereby allowing potential users of the component to understand its intent quickly. Assertions, known as *preconditions*, *post conditions*, and *invariants,* are added to the component specification.[9] Assertions let developers know what to expect from the component and how it behaves under certain conditions. Assertions make it easier for developers to identify qualified components, and as a consequence, be more willing to trust using the component in their designs. Design by contract is enhanced when components have an "economical interface," that is, the component interface has the smallest set of operations necessary to allow it to fulfill its responsibilities (contract).

An interface specification provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application. Among the many factors considered during component qualification are [Bro96]:

? What factors are considered during component qualification?

- Application programming interface (API).
- Development and integration tools required by the component.
- Run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol.

9 *Preconditions* are statements about assumptions that must be verified before using a component, *post conditions* statements about guaranteed services a component will deliver, and *invariants* are statements about system attributes that will not be changed by components. These concepts will be discussed in Chapter 28.

- Service requirements, including operating system interfaces and support from other components.
- Security features, including access controls and authentication protocol.
- Embedded design assumptions, including the use of specific numerical or nonnumeric algorithms.
- Exception handling.

Each of these factors is relatively easy to assess when proposing reusable components that have been developed in-house. If good software engineering practices were applied during the development of a component, answers to the questions implied by the list can be developed. However, it is much more difficult to determine the internal workings of commercial off-the-shelf (COTS) or third-party components because the only available information may be the interface specification itself.

Component Adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of "easy integration" is that consistent methods of resource management have been implemented for all components in the library, common activities such as data management exist for all components, and interfaces within the architecture and with the external environment have been implemented in a consistent manner.

ADVICE

In addition to assessing whether the cost of adaptation for reuse is justified, you should also assess whether achieving required functionality and performance can be done cost effectively.

In reality, even after a component has been qualified for use within an application architecture, conflicts may occur in one or more of the areas just noted. To avoid these conflicts, an adaptation technique called *component wrapping* [Bro96] is sometimes used. When a software team has full access to the internal design and code for a component (often not the case unless open-source COTS components are used), *white-box wrapping* is applied. Like its counterpart in software testing (Chapter 23), white-box wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. *Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box wrapping* requires the introduction of pre- and postprocessing at the component interface to remove or mask conflicts. You must determine whether the effort required to adequately wrap a component is justified or whether a custom component (designed to eliminate the conflicts encountered) should be engineered instead.

Component Composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure (usually a library of specialized components) provides a model for the coordination of

components and specific services that enable components to coordinate with one another and perform common tasks.

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia have proposed standards for component software.[10] These standards include: CCM (Corba Component Model),[11] Microsoft COM and .NET,[12] JavaBeans,[13] and OSGI (Open Services Gateway Initiative [OSG13].[14] None of these standards dominate the industry. Although many developers have standardized on one, it is likely that large software organizations may choose to use a standard based on the application categories and platforms that are chosen.

14.7.3 Architectural Mismatch

One of the challenges facing widespread reuse is *architectural mismatch* [Gar09a]. The designers of reusable components often make implicit assumptions about the environment to which the component is coupled. These assumptions often focus on the component control model, the nature of the component connections (interfaces), the architectural infrastructure itself, and the nature of the construction process. If these assumptions are incorrect, architectural mismatch occurs.

Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods (Chapter 28), all contribute to the creation of software components that are reusable and prevent architectural mismatch.

Early detection of architectural mismatch can occur if stakeholder assumptions are explicitly documented. In addition, the use of a risk-driven process model emphasizes the definition of early architectural prototypes and points to areas of mismatch. Repairing architectural mismatch is often very difficult without making use of mechanisms like wrappers or adapters.[15] Sometimes it is necessary to completely redesign a component interface or the component itself to remove coupling issues.

---

10 Greg Olsen [Ols06] provides an excellent discussion of past and present industry efforts to make CBSE a reality. Ivica Crnkovic [Crb11] presents a discussion of more recent industrial component models.

11 Further information on CCM can be found at:**www.omg.org**

12 Information on COM and .Net can be found at: **www.microsoft.com/COM** and **msdn2.microsoft.com/en-us/netframework/default.aspx**

13 The latest information on Javabeans can be found at: **java.sun.com/products/javabeans/docs/**

14 Information on OSGI can be found at: **http://www.osgi.org/Main/HomePage**

15 An *adapter* is a software device that allows a client with an incompatible interface to access a component by translating a request for service into a form that can access the original interface.

14.7.4 Analysis and Design for Reuse

Elements of the requirements model (Chapters 9–11) are compared to descriptions of reusable components in a process that is sometimes referred to as "specification matching" [Bel95]. If specification matching points to an existing component that fits the needs of the current application, you can extract the component from a reuse library (repository) and use it in the design of a new system. If components cannot be found (i.e., there is no match), a new component is created. It is at this point—when you begin to create a new component—that *design for reuse* (DFR) should be considered.

ADVICE

DFR can be quite difficult when components must be interfaced or integrated with legacy systems or with multiple systems whose architecture and interfacing protocols are inconsistent.

As we have already noted, DFR requires that you apply solid software design concepts and principles (Chapter 12). But the characteristics of the application domain must also be considered. Binder [Bin93] suggests a number of key issues[16] that form a basis for design for reuse. If the application domain has standard global data structures, the component should be designed to make use of these standard data structures. Standard interface protocols within an application domain should be adopted, and an architectural style (Chapter 13) that is appropriate for the domain can serve as a template for the architectural design of new software. Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

14.7.5 Classifying and Retrieving Components

Consider a large component repository. Tens of thousands of reusable software components reside in it. But how do you find the one that you need? To answer this question, another question arises: How do we describe software components in unambiguous, classifiable terms? These are difficult questions, and no definitive answer has yet been developed.

A reusable software component can be described in many ways, but an ideal description encompasses what Tracz [Tra95] has called the *3C model*—concept, content, and context—a description of what the component accomplishes, how this is achieved with content that may be hidden from casual users and need be known only to those who intend to modify or test the component, and where the component resides within its domain of applicability.

To be of use in a pragmatic setting, concept, content, and context must be translated into a concrete specification scheme. Dozens of papers and articles have been written about classification schemes for reusable software components (e.g., [Nir10], [Cec06]), and all should be implemented within a reuse environment that exhibits the following characteristics:

16 In general, DFR preparations should be undertaken as part of domain engineering.

? What are the key characteristics of a component reuse environment?

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

Each of these functions interacts with or is embodied within the confines of a reuse library, one element of a larger software repository (Chapter 29) that provides facilities for the storage of software components and a wide variety of reusable work products (e.g., specifications, designs, patterns, frameworks, code fragments, test cases, user guides).

SOFTWARE TOOLS

CBSE

Objective: To aid in modeling, design, review, and integration of software components as part of a larger system.

Mechanics: Tools mechanics vary. In general, CBSE tools assist in one or more of the following capabilities: specification and modeling of the software architecture, browsing and selection of available software components; integration of components.

Representative Tools[17]

Component Source (**www.componentsource.com**) provides a wide array of COTS software components (and tools) supported within many different component standards.

Component Manager, developed by Flashline (**http://www.softlookup.com/download.asp?id=8204**), "is an application that enables, promotes, and measures software component reuse."

Select Component Factory, developed by Select Business Solutions (**www.selectbs.com**), "is an integrated set of products for software design, design review, service/component management, requirements management and code generation."

Software Through Pictures-ACD, distributed by Aonix (**www.aonix.com**), enables comprehensive modeling using UML for the OMG model-driven architecture—an open, vendor-neutral approach for CBSE.

14.8 SUMMARY

The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Three different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view

17 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The traditional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

A number of important principles and concepts guide the designer as classes are elaborated. Ideas encompassed in the Open-Closed Principle and the Dependency Inversion Principle and concepts such as coupling and cohesion guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated by specifying messaging details, identifying appropriate interfaces, elaborating attributes and defining data structures to implement them, describing processing flow within each operation, and representing behavior at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Traditional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Component-level design for WebApps considers both content and functionality as it is delivered by a Web-based system. Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. Functional design for WebApps focuses on processing functions that manipulate content, perform computations, query and access a database, and establish interfaces with other systems. All component-level design principles and guidelines apply.

Component-level design for mobile apps makes use of a multilayered architecture that includes a user interface layer, a business layer, and a data layer. If the mobile app requires the design of components that implement the business and/or data layers on the mobile device, the limitations of the physical characteristics of the device become important constraints on the design.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Component-based software engineering identifies, constructs, catalogs, and disseminates a set of software components in a particular application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

Problems and Points to Ponder

14.1. The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for object-oriented and traditional software. Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

14.2. Why are control components necessary in traditional software and generally not required in object-oriented software?

14.3. Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?

14.4. Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?

14.5. Select three components that you have developed recently and assess the types of cohesion that each exhibits. If you had to define the primary benefit of high cohesion, what would it be?

14.6. Select three components that you have developed recently and assess the types of coupling that each exhibits. If you had to define the primary benefit of low coupling, what would it be?

14.7. Is it reasonable to say that problem domain components should never exhibit external coupling? If you agree, what types of component would exhibit external coupling?

14.8. Develop (1) an elaborated design class, (2) interface descriptions, (3) an activity diagram for one of the operations within the class, and (4) a detailed statechart diagram for one of the *SafeHome* classes that we have discussed in earlier chapters.

14.9. Are stepwise refinement and refactoring the same thing? If not, how do they differ?

14.10. What is a WebApp component?

14.11. Select a small portion of an existing program (approximately 50 to 75 source lines). Isolate the structured programming constructs by drawing boxes around them in the source code. Does the program excerpt have constructs that violate the structured programming philosophy? If so, redesign the code to make it conform to structured programming constructs. If not, what do you notice about the boxes that you've drawn?

14.12. All modern programming languages implement the structured programming constructs. Provide examples from three programming languages.

14.13. Select a small coded component and represent it using an activity diagram.

14.14. Why is "chunking" important during the component-level design review process?

Further Readings and Information Sources

Many books on component-based development and component reuse have been published in recent years. Szyperski (*Component Software,* 2nd ed., Addison-Wesley, 2011) emphasizes the importance of software components as building blocks for effective systems. Hamlet, (*Composing Software Components*, Springer, 2010), Curtis (*Modular Web Design*, New Riders, 2009), Apperly and his colleagues (*Service- and Component-Based Development,* Addison-Wesley, 2004), Heineman and Councill (*Component Based Software Engineering,* Addison-Wesley, 2001), Brown (*Large-Scale Component-Based Development,* Prentice Hall, 2000), Allen (*Realizing e-Business with Components,* Addison-Wesley, 2000), and Leavens and Sitaraman (*Foundations of Component-Based Systems,* Cambridge University Press, 2000) cover many important aspects of the CBSE process. Stevens (*UML Components*, Addison-Wesley, 2006), Apperly and his colleagues (*Service- and Component-Based*

Development, 2nd ed., Addison-Wesley, 2003), Cheesman and Daniels (*UML Components,* Addison-Wesley, 2000) discussed CBSE with a UML emphasis.

Malik (*Component-Based Software Development,* Lap Lambert Publishing, 2013) presents methods for building effective component repositories. Gross (*Component-Based Software Testing with UML,* Springer, 2010) and Gao and his colleagues (*Testing and Quality Assurance for Component-Based Software,* Artech House, 2006) discuss testing and SQA issues for component-based systems.

Dozens of books describing the industry's component-based standards have been published in recent years. These address the inner workings of the standards themselves but also consider many important CBSE topics.

The work of Linger, Mills, and Witt (*Structured Programming—Theory and Practice,* Addison-Wesley, 1979) remains a definitive treatment of the subject. The text contains a good PDL as well as detailed discussions of the ramifications of structured programming. Other books that focus on procedural design issues for traditional systems include those by Farrell (*A Guide to Programming Logic and Design,* Course Technology, 2010), Robertson (*Simple Program Design,* 5th ed., Course Technology, 2006), Bentley (*Programming Pearls,* 2nd ed., Addison-Wesley, 1999), and Dahl (*Structured Programming,* Academic Press, 1997).

Relatively few recent books have been dedicated solely to component-level design. In general, programming language books address procedural design in some detail but always in the context of the language that is introduced by the book. Hundreds of titles are available.

A wide variety of information sources on component-level design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to component-level design can be found at the SEPA website: **www.mhhe.com/pressman**

CHAPTER

User Interface Design

15

Key Concepts

accessibility 336
command labeling . 335
control 318
design evaluation .. 342
error handling 335
golden rules 318
help facilities 335
interface analysis . 325
interface consistent 321
interface design .. 332
interface design models 322
internationalization . 336
memory load 319
principles and guidelines 337
process 323
response time 335
task analysis 326
task elaboration .. 327
usability 322
user analysis 325
webApp and mobile interface design .. 337

We live in a world of high-technology products, and virtually all of them—consumer electronics, industrial equipment, automobiles, corporate systems, military systems, personal computer software, mobile apps, and WebApps—require human interaction. If a product is to be successful, it must exhibit good *usability*—a qualitative measure of the ease and efficiency with which a human can employ the functions and features offered by the high-technology product.

For the first three decades of the computing era, usability was not a dominant concern among those who built software. In his classic book on design, Donald Norman [Nor88] argued that it was time for a change in attitude:

> To make technology that fits human beings, it is necessary to study human beings. But now we tend to study only the technology. As a result, people are required to conform to technology. It is time to reverse this trend, time to make technology that conforms to people.

As technologists studied human interaction, two dominant issues arose. First, a set of *golden rules* (discussed in Section 15.1) were identified. These applied to all human interaction with technology products. Second, a set of *interaction mechanisms* were defined to enable software designers to build systems that properly implemented the golden rules. These interaction mechanisms, collectively called the user interface, have eliminated some of the most egregious problems associated with human interfaces. But even today, we all encounter user interfaces that are difficult to learn, difficult to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

Quick Look

What is it? User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling of windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

How do I ensure that I've done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

15.1 The Golden Rules

In his book on interface design, Theo Mandel [Man97] coins three *golden rules:*

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

15.1.1 Place the User in Control

Quote:

"It's better to design the user experience than rectify it."

Jon Meads

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.

"What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that."

Your first reaction might be to shake your head and smile, but pause for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface

that is easy to build, but frustrating to use. Mandel [Man97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to drag a document into the "trash" is an implementation of direct manipulation.

Quote:

"I have always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer know how to use my telephone."

Bjarne Stronstrup (originator of C++)

15.1.2 Reduce the User's Memory Load

A well-designed user interface does not tax a user's memory because the more a user has to remember, the more error-prone the interaction will be. Whenever possible, the system should "remember" pertinent information and assist

the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the *print* function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest.

SafeHome

Violating a UI Golden Rule

The scene: Vinod's cubicle, as user interface design begins.

The players: Vinod and Jamie, members of the *SafeHome* software engineering team.

The conversation:

Jamie: I've been thinking about the surveillance function interface.

Vinod (smiling): Thinking is good.

Jamie: I think maybe we can simplify matters some.

Vinod: Meaning?

Jamie: Well, what if we eliminate the floor plan entirely. It's flashy, but it's going to take serious development effort. Instead we just ask the user to specify the camera he wants to see and then display the video in a video window.

Vinod: How does the homeowner remember how many cameras are set up and where they are?

Jamie (mildly irritated): He's the homeowner; he should know.

Vinod: But what if he doesn't?

Jamie: He should.

Vinod: That's not the point . . . what if he forgets?

Jamie: Uh, we could provide a list of operational cameras and their locations.

Vinod: That's possible, but why should he have to ask for a list?

Jamie: Okay, we provide the list whether he asks or not.

Vinod: Better. At least he doesn't have to remember stuff that we can give him.

Jamie (thinking for a moment): But you like the floor plan, don't you?

Vinod: Uh huh.

Jamie: Which one will marketing like, do you think?

Vinod: You're kidding, right?

Jamie: No.

Vinod: Duh … the one with the flash … they love sexy product features … they're not interested in which is easier to build.

Jamie (sighing): Okay, maybe I'll prototype both.

Vinod: Good idea … then we let the customer decide.

Quote:

"Things that look different should act different. Things that look the same should act the same."

Larry Marine

15.1.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a complete product line. A family of applications (i.e., a product line) should implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application encountered. A change (e.g., using alt-S to invoke scaling) will cause confusion.

The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the interface design process itself.

INFO

Usability

In an insightful paper on usability, Larry Constantine [Con95] asks a question that has significant bearing on the subject: "What do users want, anyway?" He answers this way:

> "What users really want are good tools. All software systems, from operating systems and languages to data entry and decision support applications, are just tools. End users want from the tools we engineer for them much the same as we expect from the tools we use. They want systems that are easy to learn and that help them do their work. They want software that doesn't slow them down, that doesn't trick or confuse them, that does make it easier to make mistakes or harder to finish the job."

Constantine argues that usability is not derived from aesthetics, state-of-the-art interaction mechanisms, or built-in interface intelligence. Rather, it occurs when the architecture of the interface fits the needs of the people who will be using it.

A formal definition of usability is somewhat illusive. Donahue and his colleagues [Don99] define it in the following manner: "Usability is a measure of how well a computer system … facilitates learning; helps learners remember what they've learned; reduces the likelihood of errors; enables them to be efficient, and makes them satisfied with the system."

The only way to determine whether "usability" exists within a system you are building is to conduct usability assessment or testing. Watch users interact with the system and answer the following questions [Con95]:

- Is the system usable without continual help or instruction?
- Do the rules of interaction help a knowledgeable user to work efficiently?
- Do interaction mechanisms become more flexible as users become more knowledgeable?
- Has the system been tuned to the physical and social environment in which it will be used?
- Is the user aware of the state of the system? Does the user know where she is at all times?
- Is the interface structured in a logical and consistent manner?
- Are interaction mechanisms, icons, and procedures consistent across the interface?
- Does the interaction anticipate errors and help the user correct them?
- Is the interface tolerant of errors that are made?
- Is the interaction simple?

If each of these questions is answered yes, it is likely that usability has been achieved.

Among the many measurable benefits derived from a usable system are [Don99]: increased sales and customer satisfaction, competitive advantage, better reviews in the media, better word of mouth, reduced support costs, improved end-user productivity, reduced training costs, reduced documentation costs, reduced likelihood of litigation from unhappy customers.

15.2 User Interface Analysis and Design

WebRef

An excellent source of UI design information can be found at **www.nngroup.com**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

15.2.1 Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model*, the software engineer creates a *design model*, the end user develops a mental image that is often called the user's *mental model* or the *system perception*, and the

Quote:

"If there's a 'trick' to it, the UI is broken."

Douglas Anderson

implementers of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. In his introductory column on "user-centric design," Jeff Patton [Pat07] notes:

> The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Self-substitution isn't user centric—it's self-centric.

ADVICE

Even a novice user wants shortcuts; even knowledgeable, frequent users sometimes need guidance. Give them what they need.

To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [Shn04]. In addition, users can be categorized as novices, knowledgeable, intermittent users, or knowledgeable frequent users.

The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a mobile app that rates restaurants were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend on the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands restaurant rating apps fully but has worked with the specific app only a few times might actually be able to provide a more complete description of its function than the novice who has spent days trying to apply the app effectively.

Quote:

"[P]ay attention to what users do, not what they say."

Jakob Nielsen

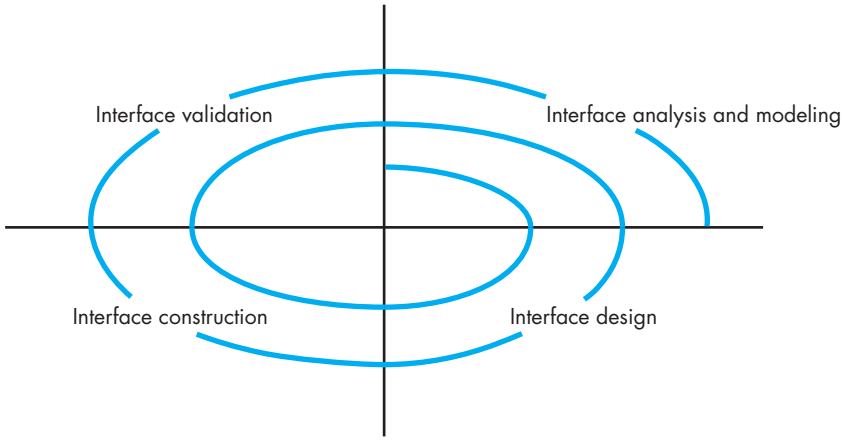
The *implementation model* combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

15.2.2 The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 4. Referring to Figure 15.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities [Man97]: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 15.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

FIGURE 15.1

The user interface design process



Quote:

"It's better to design the user experience than rectify it."

Jon Meads

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception (Section 15.2.1) for each class of users.

Once general requirements have been defined, a more detailed *task analysis* is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 15.3. Finally, analysis of the user environment focuses on the characteristics of the physical work environment (e.g., location, lighting, position constraints).

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. Interface design is discussed in more detail in Section 15.4.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 15.5) may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the user's acceptance of the interface as a useful tool in his or her work.

As we have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis

or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

15.3 Interface Analysis[1]

A key tenet of all software engineering process models is this: *understand the problem before you attempt to design a solution.* In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. In the sections that follow, we examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

15.3.1 User Analysis

The phrase *user interface* is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier we noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources (user interviews, sales input, marketing input, support input) can be used to accomplish this.

The following set of questions (adapted from [Hac98]) will help you to better understand the users of a system:

KEY POINT

How do we learn about the demographics and characteristics of end users?

- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?

1 It is reasonable to argue that this section should be placed in Chapter 8, 9, 10, or 11, since requirements analysis issues are discussed there. It has been positioned here because interface analysis and design are intimately connected to one another, and the boundary between the two is often fuzzy.

- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

15.3.2 Task Analysis and Modeling

The goal of task analysis is to answer the following questions:

KEY POINT

The user's goal is to accomplish one or more tasks via the UI. To accomplish this, the UI must provide mechanisms that allow the user to achieve her goal.

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

To answer these questions, you must draw upon techniques that we have discussed earlier in this book, but in this instance, these techniques are applied to the user interface.

WebRef

An excellent source of information on user modeling can be found at **http://web.eecs.umich.edu/~kieras/docs/GOMS/**

Use Cases. In previous chapters you learned that the use case describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function. When asked: "How do you decide where to put furniture in a room?" an interior designer writes the following informal use case:

> I begin by sketching the floor plan of the room, the dimensions and the location of windows and doors. I'm very concerned about light as it enters the room, about the view out of the windows (if it's beautiful, I want to draw attention to it), about the running length of an unobstructed wall, about the flow of movement through the room. I then look at the list of furniture my customer and I have chosen. . . . Then, I draw a rendering (a 3-D picture) of the room to give my customer a feel for what it'll look like.

SafeHome

Use Cases for UI Design

The scene: Vinod's cubicle, as user interface design continues.

The players: Vinod and Jamie, members of the *SafeHome* software engineering team.

The conversation:

Jamie: I pinned down our marketing contact and had her write a use case for the surveillance interface.

Vinod: From whose point of view?

Jamie: The homeowner, who else is there?

Vinod: There's also the system administrator role, even if it's the homeowner playing the role, it's a different point of view. The administrator sets the system up, configures stuff, lays out the floor plan, places the cameras . . .

Jamie: All I had her do was play the role of the homeowner when he wants to see video.

Vinod: That's okay. It's one of the major behaviors of the surveillance function interface. But we're going to have to examine the system administration behavior as well.

Jamie (irritated): You're right.

[Jamie leaves to find the marketing person. She returns a few hours later.]

Jamie: I was lucky, I found her and we worked through the administrator use case together. Basically, we're going to define "administration" as one function that's applicable to all other *SafeHome* functions. Here's what we came up with.

[Jamie shows the informal use case to Vinod.]

Informal use case: I want to be able to set or edit the system layout at any time. When I set up the system, I select an administration function. It asks me whether I want to do a new setup or whether I want to edit an existing setup. If I select a new setup, the system displays a drawing screen that will enable me to draw the floor plan onto a grid. There will be icons for walls, windows, and doors so that drawing is easy. I just stretch the icons to their appropriate lengths. The system will display the lengths in feet or meters (I can select the measurement system). I can select from a library of sensors and cameras and place them on the floor plan. I get to label each, or the system will do automatic labeling. I can establish settings for sensors and cameras from appropriate menus. If I select edit, I can move sensors or cameras, add new ones or delete existing ones, edit the floor plan, and edit the setting for cameras and sensors. In every case, I expect the system to do consistency checking and to help me avoid mistakes.

Vinod (after reading the scenario): Okay, there are probably some useful design patterns [Chapter 12] or reusable components for GUIs for drawing programs. I'll betcha 50 bucks we can implement some or most of the administrator interface using them.

Jamie: Agreed. I'll check it out.

This use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction. In addition, other features of the system that would please the interior designer might also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through each window.

Task Elaboration. In Chapter 12, we discussed stepwise elaboration (also called functional decomposition or stepwise refinement) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach that assists in understanding the human activities the user interface must accommodate.

First, you should define and classify the human tasks that are required to accomplish the goal of the system or app. For example, let's reconsider the

computer-aided design system for interior designers discussed earlier. By observing an interior designer at work, you notice that interior design comprises a number of major activities: furniture layout (note the use case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions, (2) place windows and doors at appropriate locations, (3a) use furniture templates to draw scaled furniture outlines on the floor plan, (3b) use accents templates to draw scaled accents on the floor plan, (4) move furniture outlines and accent outlines to get the best placement, (5) label all furniture and accent outlines, (6) draw dimensions to show location, and (7) draw a perspective-rendering view for the customer. A similar approach could be used for each of the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.[2] The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

ADVICE

Although object elaboration is useful, it should not be used as a stand-alone approach. The user's voice must be considered during task analysis.

Object Elaboration. Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provides a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include size, shape, location and others. The interior designer would *select* the object from the **Furniture** class, *move* it to a position on the floor plan (another object in this context), *draw* the furniture outline, and so forth. The tasks *select, move,* and *draw* are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

Workflow Analysis. When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis*. This technique allows you to understand how a work process is completed when several people

2 However, this may not be the case. The interior designer might want to specify the perspective to be drawn, the scaling, the use of color, and other information. The use case related to drawing perspective renderings would provide the information you need to address this task.

(and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process[3] will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).

We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 15.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted earlier. This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

Quote:

"It is far better to adapt the technology to the user than to force the user to adapt to the technology."

Larry Marine

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., *Fills prescription* could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical Representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the user task *requests that a prescription be refilled.* The following task hierarchy is developed:

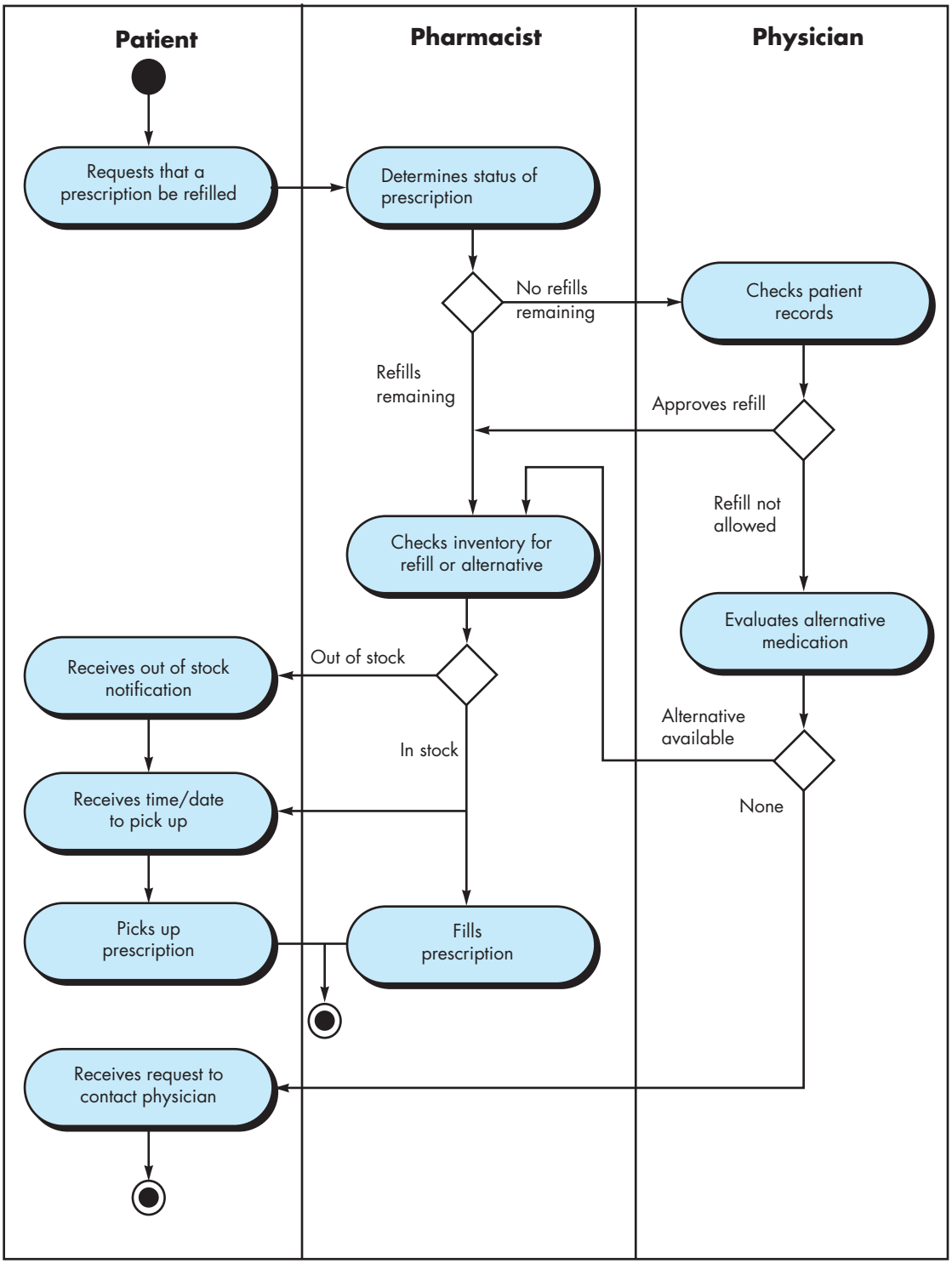
Requests that a prescription be refilled

- *Provide identifying information.*
  - *Specify name.*
  - *Specify userid.*
  - *Specify PIN and password.*
- *Specify prescription number.*
- *Specify date refill is required.*

To complete the task, three subtasks are defined. One of these subtasks, *provide identifying information,* is further elaborated in three additional sub-subtasks.

3 This example has been adapted from [Hac98].

FIGURE 15.2 Swimlane diagram for prescription refill function



15.3.3 Analysis of Display Content

The user tasks identified in Section 15.3.2 lead to the presentation of a variety of different types of content. Analysis modeling techniques discussed in Chapters 9 through 11 identify the output data objects that are produced by an application. These data objects may be (1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

? How do we determine the format and aesthetics of content displayed as part of the UI?

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?

The answers to these (and other) questions will help you to establish requirements for content presentation.

15.3.4 Analysis of the Work Environment

Hackos and Redish [Hac98] make the following statement about work environment analysis when they state: "People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people." In some applications the user interface for a computer-based system is placed in a "user-friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse or touch screen may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to

share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

15.4 Interface Design Steps

Quote:

"Interactive design [is] a seamless blend of graphic arts, technology, and psychology."

Brad Wieners

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

Although many different user interface design models (e.g., [Nor86], [Nie00]) have been proposed, all suggest some combination of the following steps: (1) define interface objects and actions (operations), (2) identify events (user actions) that will cause the state of the user interface to change, (3) depict the representation of each state, and (4) indicate how the user interprets each state from information provided through the interface.

15.4.1 Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed in much the same way as described in Chapter 9. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication of this action is to create a hard-copy report. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted. If a real-world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

To provide a brief illustration of the design steps noted previously, consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary use case (written by the homeowner) for the interface follows:

Preliminary use case: I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my notebook

computer (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming *SafeHome*. I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this use case, the following homeowner tasks, objects, and data items are identified:

- *Accesses* the *SafeHome* system
- *Enters* an **ID** and **password** to allow remote access
- *Checks* **system status**
- *Arms* or *disarms SafeHome* system
- *Displays* **floor plan** and **sensor locations**
- *Displays* **zones** on floor plan
- *Changes* **zones** on floor plan
- *Displays* **video camera locations** on floor plan
- *Selects* **video camera** for viewing
- *Views* **video images** (four frames per second)
- *Pans* or *zooms* the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

ADVICE

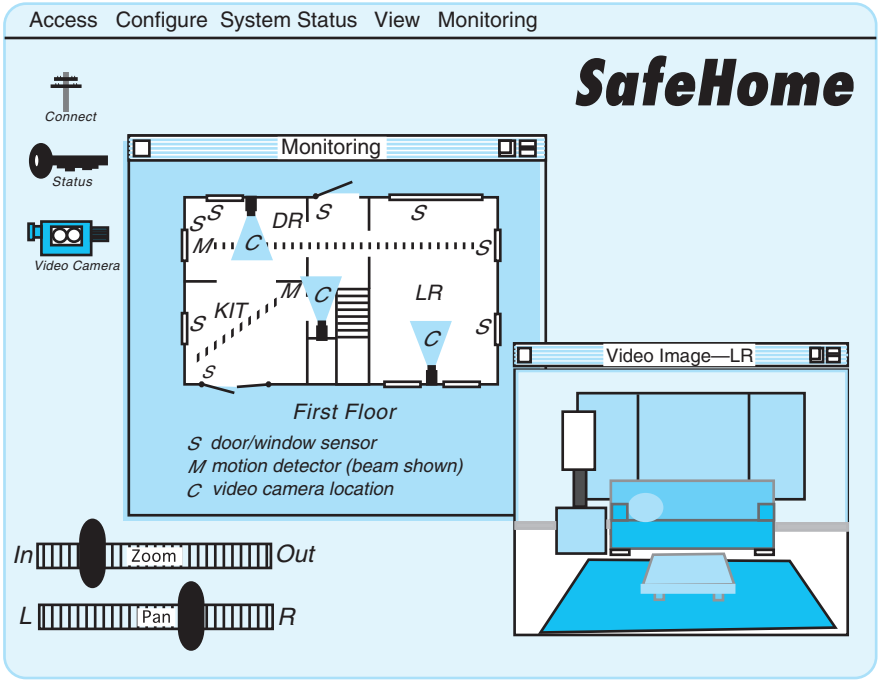
Although automated tools can be useful in developing layout prototypes, sometimes a pencil and paper are all that are needed.

A preliminary sketch of the screen layout for video monitoring is created (Figure 15.3).[4] To invoke the video image, a video camera location icon, *C*, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply

4 Note that this differs somewhat from the implementation of these features in earlier chapters. This might be considered a first draft design and represents one alternative that might be considered.

FIGURE 15.3

Preliminary screen layout



drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

15.4.2 User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. A *design pattern* is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor

is well known to every user and provides an effective mechanism for placing a future date in context.

A vast array of interface design patterns has been proposed over the past decade. A more detailed discussion of user interface design patterns is presented in Chapter 16. In addition, Erickson [Eri08] provides pointers to many Web-based collections.

15.4.3 Design Issues

WebRef

A wide variety of UI design patterns has been proposed. For pointers to a variety of patterns sites, visit **http://www.hcipatterns.org/patterns/borchers/patternIndex.html**.

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

Quote:

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."

Douglas Adams

Response Time. System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. *Variability* refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.

Help Facilities. Almost every user of an interactive, computer-based system requires help now and then. Modern software should provide online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

Quote:

"The interface from hell—'to correct this error and continue, enter any 11-digit prime number . . .'"

Author unknown

Error Handling. In general, every error message or warning produced by an interactive system should have the following characteristics: (1) describes the problem in jargon that the user can understand; (2) provides constructive advice for recovering from the error; (3) indicates any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have); be accompanied by an audible or visual cue; and should never place blame for the error on the user.

Menu and Command Labeling. The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and-pick interfaces has reduced reliance on typed commands, but some power-users

continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?
- Have appropriate conventions for command usage been established across a family of applications?

WebRef

Guidelines for developing accessible software can be found at **http://www-03.ibm.com/able/guidelines/software/access-software.html**.

Application Accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines (e.g., [W3C03])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [App13], [Mic13]) provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. *Localization* features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM13]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

SOFTWARE TOOLS

User Interface Development

Objective: These tools enable a software engineer to create a sophisticated GUI with relatively little custom software development. The tools provide access to reusable components and make the creation of an interface a matter of selecting from predefined capabilities that are assembled using the tool.

Mechanics: Modern user interfaces are constructed using a set of reusable components that are coupled with some custom components developed to provide specialized features. Most user interface development tools enable a software engineer to create an interface using "drag and drop" capability. That is, the developer selects from many predefined capabilities (e.g., forms builders, interaction mechanisms, command processing capability) and places these capabilities within the content of the interface to be created.

Representative Tools:[5]

LegaSuite GUI, developed by Seagull Software (**http://www-304.ibm.com/partnerworld/gsd/solutiondetails.do?solution=1020&expand=true&lc=en**), enables the creation of browser-based GUIs and provides facilities for reengineering antiquated interfaces.

Motif Common Desktop Environment, developed by The Open Group (**www.osf.org/tech/desktop/cde/**), is an integrated graphical user interface for open systems desktop computing. It delivers a single, standard graphical interface for the management of data and files (the graphical desktop) and applications.

Altia Design 8.0, developed by Altia (**www.altia.com**), is a tool for creating GUIs on a variety of different platforms (e.g., automotive, handheld, industrial).

15.5 WEBAPP AND MOBILE INTERFACE DESIGN

Quote:

"If a site is perfectly usable but it lacks an elegant and appropriate design style, it will fail."

Curt Cloninger

Every user interface—whether it is designed for a WebApp, a mobile device, a traditional software application, a consumer product, or an industrial device—should exhibit the usability characteristics that were discussed earlier in this chapter. Dix [Dix99] argues that WebApp and mobile interfaces should answer three primary questions: *Where am I? What can I do now? Where have I been and where can I go?* Answers to these questions allow a user to understand context and navigate more effectively through the app.

15.5.1 Interface Design Principles and Guidelines

The user interface of a Web or mobile app is its "first impression." Regardless of the value of its content, the sophistication of its processing capabilities and services, and the overall benefit of the application itself, a poorly designed interface will disappoint the potential user and may, in fact, cause the user to go elsewhere. Because of the sheer volume of competing WebApps and mobile apps in virtually every subject area, the interface must "grab" a potential user immediately.

There are, of course, important differences between WebApps and mobile apps. By virtue of the physical constraints imposed by small mobile devices

5 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category.

(e.g., smart phones), the mobile interface designer must compress interaction in a focused manner. However, the basic principles discussed in this section continue to apply.

Bruce Tognozzi [Tog01] defines a set of fundamental design principles that lead to better usability:[6]

? Is there a set of basic principles that can be applied as you design a GUI?

Anticipation. *An application should be designed so that it anticipates the user's next move.* For example, a user has requested a content object that presents information about a printer driver for a new version of an operating system. The designer of the WebApp should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen directly.

Communication. *The interface should communicate the status of any activity initiated by the user.* Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way).

Consistency. *The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout.* For example, if a mobile app uses a set of four icons (to represent major functions) across the bottom of the display, these icons should appear on every screen and should not be moved to the top of the display. The meaning of the icons should be self-evident within the context of the app.

Controlled Autonomy. *The interface should facilitate user movement throughout the application, but it should do so in a manner that enforces navigation conventions that have been established for the application.* For example, navigation to content requiring controlled access should be controlled by userID and password, and there should be no navigation mechanism that enables a user to circumvent these controls.

Efficiency. *The design of the application and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client-server environment that executes it.* Tognozzi [Tog01] discusses this when he writes: "This simple truth is why it is so important for everyone … to appreciate the importance of making user productivity goal one and to understand the vital difference between building an efficient [application] and empowering an efficient user."

Flexibility. *The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the application in a somewhat*

6 Tognozzi's original principles have been adapted and extended for use this book. See [Tog01] for further discussion of these principles.

Quote:

"The best journey is the one with the fewest steps. Shorten the distance between the user and their goal."

Author unknown

random fashion. In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.

Focus. *The interface (and the content it presents) should stay focused on the user task(s) at hand.* This concept is particularly important for mobile apps which can become very cluttered in the designer attempts to do too much.

Human Interface Objects. *A vast library of reusable human interface objects has been developed for both Web and mobileApps. Use them.* Any interface object that can be "seen, heard, touched or otherwise perceived" [Tog01] by an end user can be acquired from any one of a number of object libraries.

Latency Reduction. *Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the application should use multitasking in a way that lets the user proceed with work as if the operation has been completed.* In addition to reducing latency, delays must be acknowledged so that the user understands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the application, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providing some entertainment (e.g., an animation or text presentation) while lengthy processing occurs.

ADVICE

Metaphors are an excellent idea because they mirror real-world experience. Just be sure that the metaphor you choose is well known to end users.

Learnability. *An application interface should be designed to minimize learning time and, once learned, to minimize relearning required when the app is revisited.* In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.

Metaphors. *An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user.* A metaphor should call on images and concepts from the user's experience, but it does not need to be an exact reproduction of a real-world experience.

Readability. *All information presented through the interface should be readable by young and old.* The interface designer should emphasize readable type styles, user-controllable font sizes, and color background choices that enhance contrast.

Track State. *When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.* In general, cookies can be designed to store state information. However, cookies are a controversial technology, and other design solutions may be more palatable for some users.

Visible Navigation. *A well-designed interface provides "the illusion that users are in the same place, with the work brought to them"* [Tog01]. When this approach is used, navigation is not a user concern. Rather, the user retrieves content object and selects functions that are displayed and executed through the interface.

SafeHome

Interface Design Review

The scene: Doug Miller's office.

The players: Doug Miller (manager of the *SafeHome* software engineering group) and Vinod Raman, a member of the *SafeHome* product software engineering team.

The conversation:

Doug: Vinod, have you and the team had a chance to review the **SafeHomeAssured.com** e-commerce interface prototype?

Vinod: Yeah . . . we all went through it from a technical point of view, and I have a bunch of notes. I e-mailed 'em to Sharon [manager of the WebApp team for the outsourcing vendor for the *SafeHome* e-commerce website] yesterday.

Doug: You and Sharon can get together and discuss the small stuff . . . give me a summary of the important issues.

Vinod: Overall, they've done a good job, nothing ground breaking, but it's a typical e-commerce interface, decent aesthetics, reasonable layout, they've hit all the important functions . . .

Doug (smiling ruefully): But?

Vinod: Well, there are a few things. . . .

Doug: Such as . . . ?

Vinod (showing Doug a sequence of storyboards for the interface prototype): Here's the major functions menu that's displayed on the home page:

Learn about *SafeHome*.
Describe your home.
Get *SafeHome* component recommendations.
Purchase a *SafeHome* system.
Get technical support.

The problem isn't with these functions. They're all okay, but the level of abstraction isn't right.

Doug: They're all major functions, aren't they?

Vinod: They are, but here's the thing . . . you can purchase a system by inputting a list of components . . . no real need to describe the house if you don't want to. I'd suggest only four menu options on the home page:

Learn about *SafeHome*.
Specify the *SafeHome* system you need.
Purchase a *SafeHome* system.
Get technical support.

When you select **Specify the *SafeHome* system you need**, you'll then have the following options:

Select *SafeHome* components.
Get *SafeHome* component recommendations.

If you're a knowledgeable user, you'll select components from a set of categorized pull-down menus for sensors, cameras, control panels, and more. If you need help, you'll ask for a recommendation and that will require that you describe your house. I think it's a bit more logical.

Doug: I agree. Have you talked with Sharon about this?

Vinod: No, I want to discuss this with marketing first; then I'll give her a call.

Quote:

"People have very little patience for poorly designed WWW sites."

Jakob Nielsen and Annette Wagner

Nielsen and Wagner [Nie96] suggest a few pragmatic "don'ts" for interface design (based on their redesign of a major WebApp). These provide a nice complement to the principles suggested earlier in this section.

- Don't force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.
- Don't make users scroll unless it is absolutely unavoidable.
- Don't rely on browser functions to assist in navigation.
- Don't allow aesthetics to supersede functionality.

- Don't force the user to search the display to determine how to link to other content or services.

A well-designed interface improves the user's perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound.

15..5.2 Interface Design Workflow for Web and Mobile Apps[7]

Earlier in this chapter we noted that user interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios (use cases) are created and analyzed to define a set of interface objects and actions.

Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are then used to prototype and ultimately implement the interface design model. The following tasks represent a rudimentary workflow:

1. **Review information contained in the requirements model and refine as required.**
2. **Develop a rough sketch of the WebApp interface layout.** If the interface layout (a prototype developed during requirements modeling) already exists, it should be reviewed and refined as required.
3. **Map user objectives into specific interface actions.** For the vast majority of Web and mobile apps, the user will have a relatively small set of primary objectives. These should be mapped into specific interface. In essence, you must answer the following question: "How does the interface enable the user to accomplish each objective?"
4. **Define a set of user tasks that are associated with each action.** Each interface action (e.g., "buy a product") is associated with a set of user tasks. These tasks have been identified during requirements modeling. During design, they must be mapped into specific interactions that encompass navigation issues, content objects, and application functions.
5. **Storyboard screen images for each interface action.** As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction. Content objects should be identified (even if they have not yet been designed and developed), and navigation links should be indicated.

7 More detailed discussions of design for WebApps and mobile applications are presented in Chapters 17 and 18, respectively.

6. **Refine interface layout and storyboards using input from aesthetic design.** In most cases, you'll be responsible for rough layout and storyboarding, but the aesthetic look and feel for a major commercial website is often developed by graphic designers, rather than technical professionals.
7. **Identify user interface objects that are required to implement the interface.** This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the interface. In addition, any custom classes are specified at this time.
8. **Develop a procedural representation of the user's interaction with the interface.** This optional task uses UML sequence diagrams and/or activity diagrams (Appendix 1) to depict the flow of activities (and decisions) that occur as the user interacts with the WebApp.
9. **Develop a behavioral representation of the interface.** This optional task makes use of UML state diagrams (Appendix 1) to represent state transitions and the events that cause them. Control mechanisms (i.e., the objects and actions available to the user to alter the state of an app) are defined.
10. **Describe the interface layout for each state.** Using design information developed in Tasks 2 and 5, associate a specific layout or screen image with each WebApp state described in Task 8.
11. **Refine and review the interface design model.** Review of the interface should focus on usability.

It is important to note that the final task set you choose should be adapted to the special requirements of the application that is to be built.

15.6 Design Evaluation

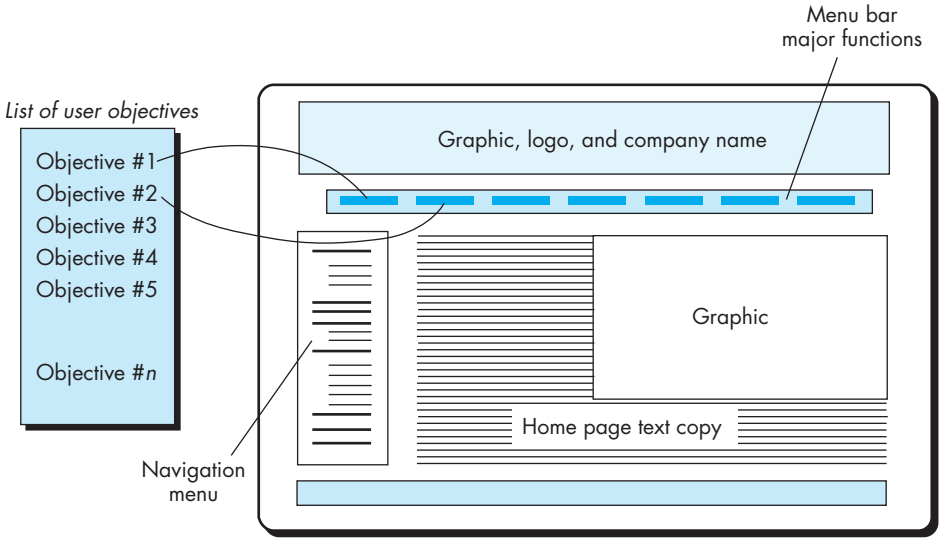
Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

The user interface evaluation cycle takes the form shown in Figure 15.4. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user,[8] who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data

8 It is important to note that experts in ergonomics and interface design may also conduct reviews of the interface. These reviews are called *heuristic evaluations* or *cognitive walkthroughs*.

FIGURE 15.4

The interface design evaluation cycle



(e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built?[9] If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria [Mor81] can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

9 Some software engineers prefer to develop a low-fidelity mockup of the user interface (UI) called a paper prototype to allow stakeholder to test the UI concept before committing any programming resources. The process is described here http://www.paperprototyping.com/what_examples.html

4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires that allow users to assess the interface prototype can be distributed. If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

A complete discussion of user interface evaluation methods is beyond the scope of this book. For further information, see [Hac98] and [Sto05].

15.7 Summary

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power and informational content of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

The development of a user interface begins with a series of analysis tasks. User analysis defines the profiles of various end users and is gathered from a variety of business and technical sources. Task analysis defines user tasks and actions using either an elaborative or object-oriented approach, applying use cases, task and object elaboration, workflow analysis, and hierarchical task representations to fully understand the human-computer interaction. Environmental analysis identifies the physical and social structures in which the interface must operate.

Once tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. This provides a basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

Like interface design for conventional software, the design of Web and mobile app interfaces describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. A set of interface design principles and an interface design workflow guide a WebApp or mobile app designer when layout and interface control mechanisms are designed.

The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system.

Problems and Points to Ponder

15.1. Describe the worst interface that you have ever worked with and critique it relative to the concepts introduced in this chapter. Describe the best interface that you have ever worked with and critique it relative to the concepts introduced in this chapter.

15.2. Develop two additional design principles that "place the user in control."

15.3. Develop two additional design principles that "reduce the user's memory load."

15.4. Develop two additional design principles that "make the interface consistent."

15.5. Consider one of the following interactive applications (or an application assigned by your instructor):

a. A desktop publishing system
b. A computer-aided design system
c. An interior design system (as described in Section 15.3.2)
d. An automated course registration system for a university
e. A library management system
f. An Internet-based polling booth for public elections
g. A home banking system
h. An interactive application assigned by your instructor

Develop a user model, design model, mental model, and an implementation model, for any one of these systems.

15.6. Perform a detailed task analysis for any one of the systems listed in Problem 15.5. Use either an elaborative or object-oriented approach.

15.7. Add at least five additional questions to the list developed for content analysis in Section 15.3.3.

15.8. Continuing Problem 15.5, define interface objects and actions for the application you have chosen. Identify each object type.

15.9. Develop a set of screen layouts with a definition of major and minor menu items for the system you chose in Problem 15.5.

15.10. Develop a set of screen layouts with a definition of major and minor menu items for the *SafeHome* system. You may elect to take a different approach than the one shown for the screen layout in Figure 15.3.

15.11. Describe your approach to user help facilities for the task analysis design model and task analysis you have performed as part of Problems 15.5, 15.7, and 15.8.

15.12. Provide a few examples that illustrate why response time variability can be an issue.

15.13. Develop an approach that would automatically integrate error messages and a user help facility. That is, the system would automatically recognize the error type and provide a help window with suggestions for correcting it. Perform a reasonably complete software design that considers appropriate data structures and algorithms.

15.14. Develop an interface evaluation questionnaire that contains 20 generic questions that would apply to most interfaces. Have 10 classmates complete the questionnaire for an interactive system that you all use. Summarize the results and report them to your class.

Further Readings and Information Sources

Although his book is not specifically about human-computer interfaces, much of what Donald Norman (*The Design of Everyday Things,* reissue edition, Basic Books, 2002) has to say about the psychology of effective design applies to the user interface. It is recommended reading for anyone who is serious about doing high-quality interface design. Weinschenk (*100 Things Every Designer Should Know About People,* New Riders, 2011) does not specifically focus on software but presents insightful discussion of user-centered design. Johnson (*Designing with the Mind in Mind,* Morgan Kaughman, 2010) uses cognitive psychology to develop rules for the design of effective interfaces.

Graphical user interfaces are ubiquitous in the modern world of computing. Whether it's an ATM, a mobile phone, an electronic dashboard in an automobile, a website, or a business application, the user interface provides a window into the software. It is for this reason that books addressing interface design abound. Among many that are worth considering are:

Ballard *(Designing the Mobile User Experience,* Wiley, 2007).

Butow (*User Interface Design for Mere Mortals,* Addison-Wesley, 2007).

Cooper and his colleagues (*About Face 3: The Essentials of Interaction Design,* 3rd ed., Wiley, 2007).

Galitz (*The Essential Guide to User Interface Design,* 3rd ed., Wiley, 2007).

Goodwin and Cooper (*Designing for the Digital Age: How to Create Human-Centered Products and Services*, Wiley, 2009).

Hartson and Pyla (*The UX Book: Process and Guidelines for Ensuring a Quality User Experience*, Morgan Kaufman, 2012).

Lehikonen and colleagues (*Personal Content Experience: Managing Digital Life in the Mobile Age,* Wiley-Interscience, 2007).

Nielsen (*Coordinating User Interfaces for Consistency,* Morgan-Kaufmann, 2006).

Pratt and Nunes (*Interactive Design,* Rockport, 2013).

Rogers and colleagues (*Interaction Design: Beyond Human-Computer Interaction*, 3rd ed., Wiley, 2011).

Shneiderman and colleagues (*Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed., Addison-Wesley, 2009).

Tidwell (*Designing Interfaces*, O'Reilly Media, 2nd ed., 2011).

Johnson (*GUI Bloopers: Common User Interface Design Don'ts and Do's,* 2nd *ed.,* Morgan-Kaufmann, 2007) and (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers,* Morgan-Kaufmann, 2000) provides useful guidance for those that learn more effectively by examining counterexamples. An enjoyable book by Cooper (*The Inmates Are Running the Asylum,* Sams Publishing, 2004) discusses why high-tech products drive us crazy and how to design ones that don't.

A wide variety of information sources on user interface design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to user interface design can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

16 Pattern-Based Design

Key Concepts

architectural patterns 359
behavioral patterns 350
component-level design patterns 360
creational patterns 350
design mistakes . . 359
design patterns . . . 348
frameworks 351
granularity 365
kinds of patterns . . 349
pattern languages . 353
pattern-organizing table 358

Each of us has encountered a design problem and silently thought: *I wonder if anyone has developed a solution for this?* The answer is almost always—*yes!* The problem is finding the solution; ensuring that it does, in fact, fit the problem you've encountered; understanding the constraints that may restrict the manner in which the solution is applied; and finally, translating the proposed solution into your design environment.

But what if the solution were codified in some manner? What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem? It turns out that software problems have been codified and described using a standardized template, and solutions to them (along with constraints) have been proposed. Called *design patterns*, this codified method for describing problems and their solution allows the software engineering community to capture design knowledge in a way that enables it to be reused.

Quick Look

What is it? Pattern-based design creates a new application by finding a set of proven solutions to a clearly delineated set of problems. Each problem and its solution is described by a design pattern that has been cataloged and vetted by other software engineers who have encountered the problem and implemented the solution while designing other applications. Each design pattern provides you with a proven approach to one part of the problem to be solved.

Who does it? A software engineer examines each problem encountered for a new application and then attempts to find a relevant solution by searching one or more patterns repositories.

Why is it important? Have you ever heard the phrase "reinventing the wheel"? It happens all the time in software development, and it's a waste of time and energy. By using existing design patterns, you can acquire a proven solution for a specific problem. As each pattern is applied, solutions are integrated and the application to be built moves closer to a complete design.

What are the steps? The requirements model is examined in order to isolate the hierarchical set of problems to be solved. The problem space is partitioned so that subsets of problems associated with specific software functions and features can be identified. Problems can also be organized by type: architectural, component-level, algorithmic, user interface, and so forth. Once a subset of problems is defined, one or more pattern repositories are searched to determine if an existing design pattern, represented at an appropriate level of abstraction, exists. Patterns that are applicable are adapted to the specific needs of the software to be built. Custom problem solving is applied in situations for which no patterns can be found.

What is the work product? A design model that depicts the architectural structure, user interface, and component-level detail is developed.

How do I ensure that I've done it right? As each design pattern is translated into some element of the design model, work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

structural patterns . 350
system of forces . . 348
user interface design patterns 362
WebApp design patterns 364

The early history of software patterns begins not with a computer scientist but a building architect, Christopher Alexander, who recognized that a recurring set of problems was encountered whenever a building was designed. He characterized these recurring problems and their solutions as *patterns,* describing them in the following manner [Ale77]:

> Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice.

Alexander's ideas were first translated into the software world in books by Gamma [Gam95], Buschmann [Bus96], and their many colleagues.[1] Today, dozens of pattern repositories exist, and pattern-based design can be applied in many different application domains.

16.1 Design Patterns

KEY POINT

Forces are those characteristics of the problem and attributes of the solution that constrain the way in which the design can be developed.

Quote:

"Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on."

Richard P. Feynman

A *design pattern* can be characterized as "a three-part rule which expresses a relation between a certain context, a problem, and a solution" [Ale79]. For software design, *context* allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment. A set of requirements, including limitations and constraints, acts as a *system of forces* that influences how the problem can be interpreted within its context and how the solution can be effectively applied.

It is reasonable to argue that most problems have multiple solutions, but that a solution is effective only if it is appropriate within the context of the existing problem. It is the system of forces that causes a designer to choose a specific solution. The intent is to provide a solution that best satisfies the system of forces, even when these forces are contradictory. Finally, every solution has consequences that may have an impact on other aspects of the software and may themselves become part of the system of forces for other problems to be solved within the larger system.

Coplien [Cop05] characterizes an effective design pattern in the following way:

- *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from

1 Earlier discussions of software patterns do exist, but these two classic books were the first cohesive treatments of the subject.

first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.

- *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

A design pattern saves you from "reinventing the wheel," or worse, inventing a "new wheel" that is slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer.

16.1.1 Kinds of Patterns

One of the reasons that software engineers are interested in (and intrigued by) design patterns is that human beings are inherently good at pattern recognition. If we weren't, we'd be frozen in space and time—unable to learn from past experience, unwilling to venture forward because of our inability to recognize situations that might lead to high risk, unhinged by a world that seems to have no regularity or logical consistency. Luckily, none of this occurs because we do recognize patterns in virtually every aspect of our lives.

In the real world, the patterns we recognize are learned over a lifetime of experience. We recognize them instantly and inherently understand what they mean and how they might be used. Some of these patterns provide us with insight into recurring phenomenon. For example, you're on your way home from work on the interstate when your navigation system (or car radio) informs you that a serious accident has occurred on the interstate in the opposing direction. You're 4 miles from the accident, but already you begin to see traffic slowing, recognizing a pattern that we'll call **RubberNecking.** People in the travel lanes moving in your direction are slowing at the sight of the accident to get a better view of what happened on the opposite side of the highway. The **RubberNecking** pattern yields remarkably predictable results (a traffic jam), but it does nothing more than describe a phenomenon. In patterns jargon, it might be called a *non-generative* pattern because it describes a context and a problem but it does not provide any clear-cut solution.

KEY POINT

A "generative" pattern describes the problem, a context, and forces, but it also describes a pragmatic solution to the problem.

When software design patterns are considered, we strive to identify and document *generative* patterns. That is, we identify a pattern that describes an important and repeatable aspect of a system and that provides us with a way to build that aspect within a system of forces that are unique to a given context. In an ideal setting, a collection of generative design patterns could be used to "generate" an application or computer-based system whose architecture enables it to adapt to change. Sometimes called *generativity,* "the successive application of

several patterns, each encapsulating its own problem and forces, unfolds a larger solution which emerges indirectly as a result of the smaller solutions" [App00].

Design patterns span a broad spectrum of abstraction and application. *Architectural patterns* describe broad-based design problems that are solved using a structural approach. *Data patterns* describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. *Component patterns* (also referred to as *design patterns*) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture. *Interface design patterns* describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. *WebApp patterns* address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. *Mobile patterns* describe commonly encountered problems when developing solutions for mobile platforms. At a lower level of abstraction, *idioms* describe how to implement all or part of a specific algorithm or data structure for a software component within the context of a specific programming language.

ADVICE

Don't force a pattern, even if it addresses the problem at hand. If the context and forces are wrong, look for another pattern.

In their seminal book on design patterns, Gamma and his colleagues[2] [Gam95] focus on three types of patterns that are particularly relevant to object-oriented design: creational patterns, structural patterns, and behavioral patterns.

? Is there a way to categorize pattern types?

Creational patterns focus on the "creation, composition, and representation" of objects and provide mechanisms that make the instantiation of objects easier within a system and enforce "constraints on the type and number of objects that can be created within a system" [Maa07]. *Structural patterns* focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. *Behavioral patterns* address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects.

INFO

Creational, Structural, and Behavioral Patterns

A wide variety of design patterns that fit into creational, structural, and behavioral categories have been proposed and can be found on the Web. Wikipedia **(http://en.wikipedia.org/wiki/Software_design_pattern)** notes the following sampling:

Creational Patterns

- **Abstract factory pattern:** centralize decision of what factory to instantiate.
- **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations.
- **Builder pattern:** separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structural Patterns

- **Adapter pattern:** "adapts" one interface for a class into one that a client expects.

2 Gamma and his colleagues [Gam95] are often referred to as the "Gang of Four" (GoF) in patterns literature.

- **Aggregate pattern:** a version of the composite pattern with methods for aggregation of children.
- **Composite pattern:** a tree structure of objects where every object has the same interface.
- **Container pattern:** create objects for the sole purpose of holding other objects and managing them.
- **Proxy pattern:** a class functioning as an interface to another thing.
- **Pipes and filters:** a chain of processes where the output of each process is the input of the next.

Behavioral Patterns

- **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects.
- **Command pattern:** Command objects encapsulate an action and its parameters.
- **Iterator pattern:** Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator pattern:** Provides a unified interface to a set of interfaces in a subsystem.
- **Visitor pattern:** A way to separate an algorithm from an object.
- **Hierarchical visitor pattern:** Provide a way to visit every node in a hierarchical data structure such as a tree.

Comprehensive descriptions of each of these patterns can be obtained via links at **www.wikipedia.org**.

16.1.2 Frameworks

KEY POINT

A *framework* is a reusable "mini-architecture" that serves as a foundation from which other design patterns can be applied.

Patterns themselves may not be sufficient to develop a complete design. In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a *framework,* for design work. That is, you can select a "*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context . . . which specifies their collaboration and use within a given domain" [Amb98].

A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain. The plug points enable you to integrate problem-specific classes or functionality within the skeleton. In an object-oriented context, a framework is a collection of cooperating classes.

Gamma and his colleagues [Gam95] describe the differences between design patterns and frameworks in the following manner:

1. *Design patterns are more abstract than frameworks.* Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly . . .
2. *Design patterns are smaller architectural elements than frameworks.* A typical framework contains several design patterns but the reverse is never true.
3. *Design patterns are less specialized than frameworks.* Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture.

In essence, the designer of a framework will argue that one reusable mini-architecture is applicable to all software to be developed within a limited domain of application. To be most effective, frameworks are applied with no changes. Additional design elements may be added, but only via the plug points that allow the designer to flesh out the framework skeleton.

16.1.3 Describing a Pattern

Pattern-based design begins with the recognition of patterns within the application you intend to build, continues with a search to determine whether others have addressed the pattern, and concludes with the application of an appropriate pattern to the problem at hand. The second of these three tasks is often the most difficult. How do you find patterns that fit your needs?

Quote:

"Patterns are half-baked—meaning you always have to finish them yourself and adapt them to your own environment."

Martin Fowler

An answer to this question must rely on effective communication of the problem the pattern addresses, the context in which the pattern resides, the system of forces that mold the context, and the solution that is proposed. To communicate this information unambiguously, a standard form or template for pattern descriptions is required. Although a number of different pattern templates have been proposed, almost all contain a major subset of the content suggested by Gamma and his colleagues [Gam95]. A simplified pattern template is shown in the sidebar.

INFO

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name

Problem—describes the problem that the pattern addresses

Motivation—provides an example of the problem

Context—describes the environment in which the problem resides including the application domain

Forces—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered

Solution—provides a detailed description of the solution proposed for the problem

Intent—describes the pattern and what it does

Collaborations—describes how other patterns contribute to the solution

Consequences—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern

Implementation—identifies special issues that should be considered when implementing the pattern

Known uses—provides examples of actual uses of the design pattern in real applications

Related patterns—cross-references related design patterns

The names of design patterns should be chosen with care. One of the key technical problems in pattern-based design is the inability to find existing patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name.

A pattern template provides a standardized means for describing a design pattern. Each of the template entries represents characteristics of the design pattern that can be searched (e.g., via a database) so that the appropriate pattern can be found.

16.1.4 Pattern Languages and Repositories

When we use the term *language,* the first thing that comes to mind is either a natural language (e.g., English, Spanish, Chinese) or a programming language (e.g., C++, Java). In both cases the language has a syntax and semantics that are used to communicate ideas or procedural instructions is an effective manner.

ADVICE

If you can't find a pattern language that addresses your problem domain, look for analogies in another set of patterns.

When the term *language* is used in the context of design patterns, it takes on a slightly different meaning. A *pattern language* encompasses a collection of patterns, each described using a standardized template (Section 16.1.3) and interrelated to show how these patterns collaborate to solve problems across an application domain.[3]

In a natural language, words are organized into sentences that impart meaning. The structure of sentences is described by the language's syntax. In a pattern language, design patterns are organized in a way that provides a "structured method of describing good design practices within a particular domain."[4]

WebRef

For a listing of useful patterns languages see **c2.com/ppr/titles.html**. Additional information can be obtained at **hillside.net/patterns/**.

In a way, a pattern language is analogous to a hypertext instruction manual for problem solving in a specific application domain. The problem domain under consideration is first described hierarchically, beginning with broad design problems associated with the domain and then refining each of the broad problems into lower levels of abstraction. In a software context, broad design problems tend to be architectural in nature and address the overall structure of the application and the data or content that serve it. Architectural problems are refined to lower levels of abstraction, leading to design patterns that solve subproblems and collaborate with one another at the component (or class) level. Rather than a sequential list of patterns, a pattern language represents an interconnected collection in which the user can begin with a broad design problem and "burrow down" to uncover specific problems and their solutions.

An extensive list of pattern languages have been proposed for software design [Hil13] which contains pointers to design patterns that are part of pattern languages in a Web-accessible patterns repositories. The repository provides an index of all design patterns and contains hypermedia links that enable the user to understand the collaborations between patterns.

---

3 Christopher Alexander originally proposed pattern languages for building architecture and urban planning. Today, pattern languages have been developed for everything from the social sciences to the software engineering process.

4 This Wikipedia description can be found at **http://en.wikipedia.org/wiki/Pattern_language**.

16.2 Pattern-Based Software Design

The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution. Throughout the design process, you should look for every opportunity to apply existing design patterns (when they meet the needs of the design) rather than creating new ones.

16.2.1 Pattern-Based Design in Context

Pattern-based design is not used in a vacuum. The concepts and techniques discussed for architectural, component-level, and user interface design (Chapters 13 through 15 are all used in conjunction with a pattern-based approach.

In Chapter 12, we noted that a set of quality guidelines and attributes serve as the basis for all software design decisions. The decisions themselves are influenced by a set of fundamental design concepts (e.g., separation of concerns, stepwise refinement, functional independence) that are achieved using heuristics that have evolved over many decades, and best practices (e.g., techniques, modeling notation) that have been proposed to make design easier to perform and more effective as a basis for construction.

The role of pattern-based design in all of this is illustrated in Figure 16.1. A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system. The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway. It may imply the design in an abstract manner, but the requirements model does little to represent the design explicitly.

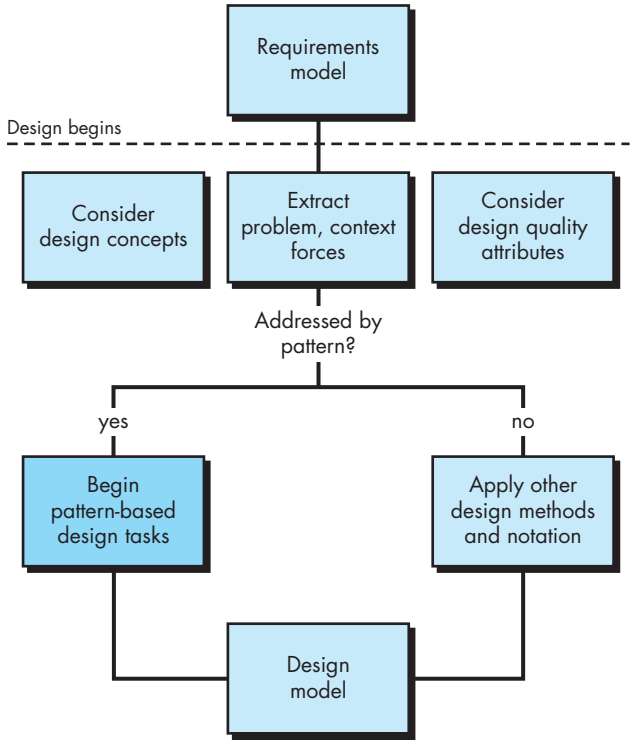
As you begin your work as a designer, it's always important to keep quality attributes (Chapter 12) in mind. These attributes establish a way to assess software quality but do little to help you actually achieve it. Therefore, you should apply proven techniques for translating the abstractions contained in the requirements model into a more concrete form that is the software design. To accomplish this, you'll use the methods and modeling tools available for architectural, component-level, and interface design. But only when you're faced with a problem, context, and system of forces that have not been solved before. If a solution already exists, use it! And that means applying a pattern-based design approach.

16.2.2 Thinking in Patterns

Pattern-based design implies with a "new way of thinking" [Sha05] that begins by considering context—the big picture. As context is evaluated, you extract a hierarchy of problems that must be solved. Some of these problems will be global in nature, while others will address specific features and functions of the software. All will be affected by a system of forces that will influence the nature of the solution that is proposed.

FIGURE 16.1

Pattern-based design in context



Shalloway and Trott [Sha05] suggest the following approach[5] that enables a designer to think in patterns:

? Pattern-based design looks interesting for the problem I have to solve. How do I get started?

1. Be sure you understand the big picture—the context in which the software to be built resides. The requirements model should communicate this to you.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with "big picture" patterns that establish a context or skeleton for further design work.
4. "Work inward from the context" [Sha05] looking for patterns at lower levels of abstraction that contribute to the design solution.
5. Repeat steps 1 to 4 until the complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.

5 Based on the work of Christopher Alexander [Ale79].

It's important to note that patterns are not independent entities. Design patterns that are present at a high level of abstraction will invariably influence the manner in which other patterns are applied at lower levels of abstraction. In addition, patterns often collaborate with one another. The implication—when you select an architectural pattern, it may very well influence the component-level design patterns you choose. Likewise, when you select a specific interface design pattern, you are sometimes forced to use other patterns that collaborate with it.

To illustrate, consider the **SafeHomeAssured.com** WebApp. If you consider the big picture, the WebApp must address how to provide information about *SafeHome* products and services, how to sell *SafeHome* products and services to customers, and how to establish Internet-based monitoring and control of an installed security system. Each of these fundamental problems can be further refined into a set of subproblems.

For example, *How to sell* via the Internet implies an **E-commerce** pattern that itself implies a large number of patterns at lower levels of abstraction. The **E-commerce** pattern (likely, an architectural pattern) implies mechanisms for setting up a customer account, displaying the products to be sold, selecting products for purchase, and so forth. Hence, if you think in patterns, it is important to determine whether a pattern for setting up an account exists. If **SetUpAccount** is available as a viable pattern for the problem context, it may collaborate with other patterns such as **BuildInputForm, ManageFormsInput,** and **ValidateFormsEntry**. Each of these patterns delineates problems to be solved and solutions that may be applied.

16.2.3 Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:

1. **Examine the requirements model and develop a problem hierarchy.** Describe each problem and subproblem by isolating the problem, the context, and the system of forces that apply. Work from broad problems (high level of abstraction) to smaller subproblems (at lower levels of abstraction).
2. **Determine if a reliable pattern language has been developed for the problem domain.** As we noted in Section 16.1.4, a pattern language addresses problems associated with a specific application domain. The *SafeHome* software team would look for a pattern language developed specifically for home security products. If that level of pattern language specificity could not be found, the team would partition the *SafeHome* software problem into a series of generic problem domains (e.g., digital device monitoring problems, user interface problems,

digital video management problems) and search for appropriate pattern languages.

3. **Beginning with a broad problem, determine whether one or more architectural patterns are available for it.** If an architectural pattern is available, be certain to examine all collaborating patterns. If the pattern is appropriate, adapt the design solution proposed and build a design model element that adequately represents it. For example, a broad problem for the **SafeHomeAssured.com** WebApp is addressed with an **E-commerce** pattern (Section 16.2.2). This pattern will suggest a specific architecture for addressing e-commerce requirements.
4. **Using the collaborations provided for the architectural pattern, examine subsystem or component-level problems and search for appropriate patterns to address them.** It may be necessary to search through other pattern repositories as well as the list of patterns that corresponds to the architectural solution. If an appropriate pattern is found, adapt the design solution proposed and build a design model element that adequately represents it. Be certain to apply step 7.
5. **Repeat steps 2 through 4 until all broad problems have been addressed.** The implication is to begin with the big picture and elaborate to solve problems at increasingly more detailed levels.
6. **If user interface design problems have been isolated (this is almost always the case), search the many user interface design pattern repositories for appropriate patterns.** Proceed in a manner similar to steps 3, 4, and 5.
7. **Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.** Be certain to examine context and forces to ensure that the pattern does, in fact, provide a solution that is amenable to the problem.
8. **Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.**

Although this design approach is top-down, real-life design solutions are sometimes more complex. Gillis [Gil06] comments on this when he writes:

> Design patterns in software engineering are meant to be used in a deductive, rationalistic fashion. So you have this general problem or requirement, X, design pattern Y solves X, therefore use Y. Now, when I reflect on my own process—and I've got reason to believe that I'm not alone here—I find that it's more organic than that, more inductive than deductive, more bottom-up than top-down.

In addition, the pattern-based approach must be used in conjunction with other software design concepts and techniques.

FIGURE 16.2

A pattern-organizing table
Source: Adapted from [Mic04].

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

16.2.4 Building a Pattern-Organizing Table

As pattern-based design proceeds, you may encounter trouble organizing and categorizing candidate patterns from multiple pattern languages and repositories. To help organize your evaluation of candidate patterns, Microsoft [Mic13] suggests the creation of a *pattern-organizing table* that takes the general form shown in Figure 16.2.

A pattern-organizing table can be implemented as a spreadsheet model using the form shown in the figure. An abbreviated list of problem statements, organized by data/content, architecture, component level, and user interface issues, is presented in the left-hand (shaded) column. Four pattern types—database, application, implementation, and infrastructure—are listed across the top row. The names of candidate patterns are noted in the cells of the table.

To provide entries for the organizing table, you'll search through pattern languages and repositories for patterns that address a particular problem statement. When one or more candidate patterns is found, it is entered in the row corresponding to the problem statement and the column that corresponds to the pattern type. The name of the pattern is entered as a hyperlink to the URL of the Web address that contains a complete description of the pattern.

16.2.5 Common Design Mistakes

ADVICE

Don't force a pattern, even if it addresses the problem at hand. If the context and forces are wrong, look for another pattern.

A number of common mistakes occur when pattern-based design is used. In some cases, not enough time has been spent to understand the underlying problem and its context and forces, and as a consequence, you select a pattern that looks right but is inappropriate for the solution required. Once the wrong pattern is selected, you refuse to see your error and force-fit the pattern. In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit. Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.

Can these mistakes be avoided? In most cases the answer is yes. Every good designer looks for a second opinion and welcomes review of her work. The review techniques discussed in Chapter 20 can help to ensure that the pattern-based design you've developed will result in a high-quality solution for the software problem to be solved.

16.3 Architectural Patterns

KEY POINT

A software architecture may have a number of architectural patterns that address issues such as concurrency, persistence, and distribution.

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed on the design.[6]

Architectural patterns are a bit different. For example, every house (and every architectural style for houses) employs a **Kitchen** pattern. The **Kitchen** pattern and patterns it collaborates with address problems associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room. In addition, the pattern might address problems associated with countertops, lighting, wall switches, a central island, flooring, and so on. Obviously, there is more than a single design for a kitchen, often dictated by the context and system of forces. But every design can be conceived within the context of the "solution" suggested by the **Kitchen** pattern.

Before a representative architectural pattern can be chosen in a particular domain, it must be assessed for its appropriateness for the application and the overall architectural style, as well as the context and system of forces that it specifies.

6 This implies that there will be a central foyer and hallway, that rooms will be placed to the left and right of the foyer, that the house will have two (or more) stories, that the bedrooms of the house will be upstairs, and so on. These "rules" are imposed once the decision is made to use the center-hall colonial style.

INFO

Design Pattern Repositories

There are many sources for design patterns available on the Web. Some patterns can be obtained from individually published pattern languages, while others are available as part of a patterns portal or patterns repository. The following Web sources are worth a look:

Hillside.net **http://hillside.net/patterns/** One of the Web's most comprehensive collections of patterns and pattern languages

Portland Pattern Repository
http://c2.com/ppr/index.html Contains pointers to a wide variety of patterns resources and collections

Pattern Index
http://c2.com/cgi/wiki?PatternIndex
An "eclectic collection of patterns"

Handbook of Software Architecture
http://researcher.watson.ibm.com/researcher/view_project.php?id=3206/
Bibliographic reference to hundreds of architectural and component design patterns

UI Patterns Collections

UI/HCI Patterns
http://www.hcipatterns.org/patterns/borchers/patternIndex.html

Jennifer Tidwell's UI patterns
http://designinginterfaces.com/patterns/

Mobile UI Design Patterns
http://profs.info.uaic.ro/~evalica/patterns/

Pattern Language for UI Design
www.maplefish.com/todd/papers/Experiences.html

Interaction Design Library for Games
http://www.eelke.com/files/pubs/usabilitypatternsingames.pdf

UI Design Patterns
www.cs.helsinki.fi/u/salaakso/patterns/

Specialized Design Patterns:

Aircraft Avionics
http://g.oswego.edu/dl/acs/acs/acs.html

Business Information Systems
www.objectarchitects.de/arcus/cookbook/

Distributed Processing
www.cs.wustl.edu/~schmidt/

IBM Patterns for e-Business
www-128.ibm.com/developerworks/patterns/

Yahoo! Design Pattern Library
http://developer.yahoo.com/ypatterns/

WebPatterns.org
http://www.welie.com/index.php

16.4 Component-Level Design Patterns

Component-level design patterns provide you with proven solutions that address one or more subproblems extracted from the requirements model. In many cases, design patterns of this type focus on some functional element of a system. For example, the **SafeHomeAssured.com** application must address the following design subproblem: *How do I get product specifications and related information for any SafeHome device?*

Having enunciated the subproblem that must be solved, you should now consider context and the system of forces that affect the solution. Examining the appropriate requirements model use case, you learn that the consumer uses the specification for a *SafeHome* device (e.g., a security sensor or camera) for informational purposes. However, other information that is related to the specification (e.g., pricing) may be used when e-commerce functionality is selected.

The solution to the subproblem involves a search. Since searching is a very common problem, it should come as no surprise that there are many search-related

patterns. Looking through a number of patterns repositories, you find the following patterns, along with the problem that each solves:

AdvancedSearch. Users must find a specific item in a large collection of items.

HelpWizard. Users need help on a certain topic related to the website or when they need to find a specific page within the site.

SearchArea. Users must find a page.

SearchTips. Users need to know how to control the search engine.

SearchResults. Users have to process a list of search results.

SearchBox. Users have to find an item or specific information.

For **SafeHomeAssured.com** the number of products is not particularly large, and each has a relatively simple categorization, so **AdvancedSearch** and **HelpWizard** are probably not necessary. Similarly, the search is simple enough not to require **SearchTips**. The description of **SearchBox**, however, is given (in part) as:

Search Box

(Adapted from **www.welie.com/patterns/showPattern.php?patternID=search)**

Problem: The users need to find an item or specific information.

Motivation: Any situation in which a keyword search is applied across a collection of content objects organized as Web pages.

Context: Rather than using navigation to acquire information or content, the user wants to do a direct search through content contained on multiple Web pages. Any website that already has primary navigation. User may want to search for an item in a category. User might want to further specify a query.

Forces: The website already has primary navigation. Users may want to search for an item in a category. Users might want to further specify a query using simple Boolean operators.

Solution: Offer search functionality consisting of a search label, a keyword field, a filter if applicable and a "go" button. Pressing the return key has the same function as selecting the go button. Also provide Search Tips and examples in a separate page. A link to that page is placed next to the search functionality. The edit box for the search term is large enough to accommodate three typical user queries (typically around 20 characters). If the number of filters is more than 2, use a combo box for filters selection, otherwise a radio button.

The search results are presented on a new page with a clear label containing at least "Search results" or similar. The search function is repeated in the top part of the page with the entered keywords, so that the users know what the keywords were.

The pattern description continues with other entries as described in Section 16.1.3.

The pattern goes on to describe how the search results are accessed, presented, matched, and so on. Based on this, the **SafeHomeAssured.com** team can design the components required to implement the search or (more likely) acquire existing reusable components.

SafeHome

Applying Patterns

The scene: Informal discussion during the design of a software increment that implements sensor control via the Internet for **SafeHomeAssured.com**.

The players: Jamie (responsible for design) and Vinod (**SafeHomeAssured.com** chief system architect).

The conversation:

Vinod: So how is the design of the camera control interface coming along?

Jamie: Not too bad. I've designed most of the capability to connect to the actual sensors without too many problems. I've also started thinking about the interface for the users to actually move, pan, and zoom the cameras from a remote Web page, but I'm not sure I've got it right yet.

Vinod: What have you come up with?

Jamie: Well, the requirements are that the camera control needs to be highly interactive—as the user moves the control, the camera should move as soon as possible. So, I was thinking of having a set of buttons laid out like a normal camera, but when the user clicks them, it controls the camera.

Vinod: Hmmm. Yeah, that would work, but I'm not sure it's right—for each click of a control you need to wait for the whole client-server communication to occur, and so you won't get a good sense of quick feedback.

Jamie: That's what I thought—and why I wasn't very happy with the approach, but I'm not sure how else I might do it.

Vinod: Well, why not just use the **InteractiveDeviceControl** pattern?

Jamie: Uhmmm—what's that? I haven't heard of it.

Vinod: It's basically a pattern for exactly the problem you are describing. The solution it proposes is basically to create a control connection to the server with the device, through which control commands can be sent. That way you don't need to send normal HTTP requests. And the pattern even shows how you can implement this using some simple AJAX techniques. You have some simple client-side JavaScript that communicates directly with the server and sends the commands as soon as the user does anything.

Jamie: Cool! That's just what I needed to solve this thing. Where do I find it?

Vinod: It's available in an online repository. Here's the URL.

Jamie: I'll go check it out.

Vinod: Yep—but remember to check the consequences field for the pattern. I seem to remember that there was something in there about needing to be careful about issues of security. I think it might be because you are creating a separate control channel and so bypassing the normal Web security mechanisms.

Jamie: Good point. I probably wouldn't have thought of that! Thanks.

16.5 User Interface Design Patterns

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of 10 categories of patterns as described by Tidwell [Tid02] and vanWelie [Wel01]. A few representative categories (discussed with a simple example[7]) follow:

7 An abbreviated pattern template is used here. Full pattern descriptions (along with dozens of other patterns) can be found at [Tid02] and [Wel01].

Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.

Pattern: *Top-level navigation*

Brief description: Used when a site or application implements a number of major functions. Provides a top-level menu, often coupled with a logo or identifying graphic, that enables direct navigation to any of the system's major functions.

Details: Major functions (generally limited to between four and seven function names) are listed across the top of the display (vertical column formats are also possible) in a horizontal line of text. Each name provides a link to the appropriate function or information source. Often used with the *bread crumbs* pattern discussed later.

Navigation elements: Each function/content name represents a link to the appropriate function or content.

Page layout. Address the general organization of pages (for websites) or distinct screen displays (for interactive applications).

Pattern: *Card stack*

Brief description: Used when a number of specific subfunctions or content categories related to a feature or function must be selected in random order. Provides the appearance of a stack of tabbed cards, each selectable with a mouse click and each representing specific subfunctions or content categories.

Details: Tabbed cards are a well-understood metaphor and are easy for the user to manipulate. Each tabbed card (divider) may have a slightly different format. Some may require input and have buttons or other navigation mechanisms; others may be informational. May be combined with other patterns such as *drop-down list, fill-in-the-blanks,* and others.

Navigation elements: A mouse click on a tab causes the appropriate card to appear. Navigation features within the card may also be present, but in general, these should initiate a function that is related to card data, not cause an actual link to some other display.

Forms and input. Consider a variety of design techniques for completing form-level input.

Pattern: *Fill-in-the-blanks*

Brief description: Allow alphanumeric data to be entered in a "text box."

Details: Data may be entered within a text box. In general, the data are validated and processed after some text or graphic indicator (e.g., a button containing "go," "submit," "next") is picked. In many cases this pattern can be combined with drop-down list or other patterns (e.g., SEARCH <*drop down list*> FOR <*fill-in-the-blanks* text box>).

Navigation elements: A text or graphic indicator that initiates validation and processing.

Navigation. Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.

Pattern: *Edit-in-place*

Brief description: Provide simple text editing capability for certain types of content in the location that it is displayed. No need for the user to enter a text editing function or mode explicitly.

Details: The user sees content on the display that must be changed. A mouse double-click on the content indicates to the system that editing is desired. The content is highlighted to signify that editing mode is available and the user makes appropriate changes.

Navigation elements: None.

E-commerce. Specific to websites, these patterns implement recurring elements of e-commerce applications.

Pattern: *Shopping cart*

Brief description: Provides a list of items selected for purchase.

Details: Lists item, quantity, product code, availability (in stock, out of stock), price, delivery information, shipping costs, and other relevant purchase information. Also provides ability to edit (e.g., remove, change quantity).

Navigation elements: Contains ability to proceed with shopping or go to checkout.

Each of the preceding example patterns (and all patterns within each category) would also have a complete component-level design, including design classes, attributes, operations, and interfaces.

A comprehensive discussion of user interface patterns is beyond the scope of this book. If you have further interest, see [Yah13], [UXM10], [Gub09], [Duy02], [Tid02], and [Bor01] for further information.

16.6 WebApp Design Patterns

Throughout this chapter you've learned that there are different types of patterns and many different ways they can be categorized. When you consider the design problems that must be solved when a WebApp is to be built, it's worth considering pattern categories by focusing on two dimensions: the design focus of the pattern and its level of granularity. *Design focus* identifies which aspect of the design model is relevant (e.g., information architecture, navigation, interaction). *Granularity* identifies the level of abstraction that is being considered (e.g., does

the pattern apply to the entire WebApp, to a single Web page, to a subsystem, or an individual WebApp component?).

16.6.1 Design Focus

In earlier chapters we emphasized a design progression that begins by considering architecture, component-level issues, and user interface representations. Design focus becomes "narrower" as you move further into design. The problems (and solutions) you will encounter when designing an information architecture for a WebApp are different from the problems (and solutions) that are encountered when performing interface design. Therefore, it should come as no surprise that patterns for WebApp design can be developed for different levels of design focus, so that you can address the unique problems (and related solutions) that are encountered at each level. WebApp patterns can be categorized using the following levels of design focus:

- **Information architecture patterns** relate to the overall structure of the information space, and the ways in which users will interact with the information.
- **Navigation patterns** define navigation link structures, such as hierarchies, rings, tours, and so on.
- **Interaction patterns** address how the interface informs the user of the consequences of a specific action.
- **Presentation patterns** address how to organize user interface control functions for better usability, how to show the relationship between an interface action and the content objects it affects, and how to establish effective content hierarchies.
- **Functional patterns** define the workflows, behaviors, processing, communications, and other algorithmic elements within a WebApp.

In most cases, it would be fruitless to explore the collection of **information architecture patterns** when a problem in interaction design is encountered. You would examine **interaction patterns,** because that is the design focus that is relevant to the work being performed.

16.6.2 Design Granularity

When a problem involves "big picture" issues, you should attempt to develop solutions (and use relevant patterns) that focus on the big picture. Conversely, when the focus is very narrow (e.g., uniquely selecting one item from a small set of five or fewer items), the solution (and the corresponding pattern) is targeted quite narrowly. In terms of the level of granularity, WebApp patterns follow the same levels of abstraction that were discussed earlier in this chapter.

Architectural patterns define the overall structure of the WebApp, indicate the relationships among different components or increments, and define the rules for specifying relationships among the elements (pages, packages, components, subsystems) of the architecture. Design patterns address a specific element of the WebApp design such as an aggregation of components to solve some design problem, relationships among elements on a page, or the mechanisms for effecting component-to-component communication. Component patterns relate to small-scale elements of a WebApp. Examples include individual interaction elements (e.g., radio buttons, textbooks), navigation items (e.g., how might you format links?) or functional elements (e.g., specific algorithms).

INFO

Hypermedia Design Patterns Repositories

There are a number of useful hypermedia patterns catalogs and repositories on the Internet. Hundreds of design patterns are represented in these representative sites:

InteractionPatterns by Tom Erickson
www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html

Web Design Patterns by Martijn vanWelie
www.welie.com/patterns/

Web Patterns for UI Design
http://www.onextrapixel.com/2010/11/03/15-ui-design-patterns-web-designers-should-keep-handy/

Improving Web Information Systems with Navigational Patterns
http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html

UI and Related Patterns compiled at Uzilla.net
http://uzilla.net/uzilla/blog/hci_directory/searchresultde45.html

Common Ground—A Pattern Language for HCI Design
www.mit.edu/~jtidwell/interaction_patterns.html

Patterns for Personal Websites
www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html

16.7 Patterns for Mobile Apps

By their nature, mobile applications are all about the interface. In many cases, mobile UI patterns [Mob12] are represented as a collection of "best of breed" screen images for apps in a variety of different categories. Typical examples might include:

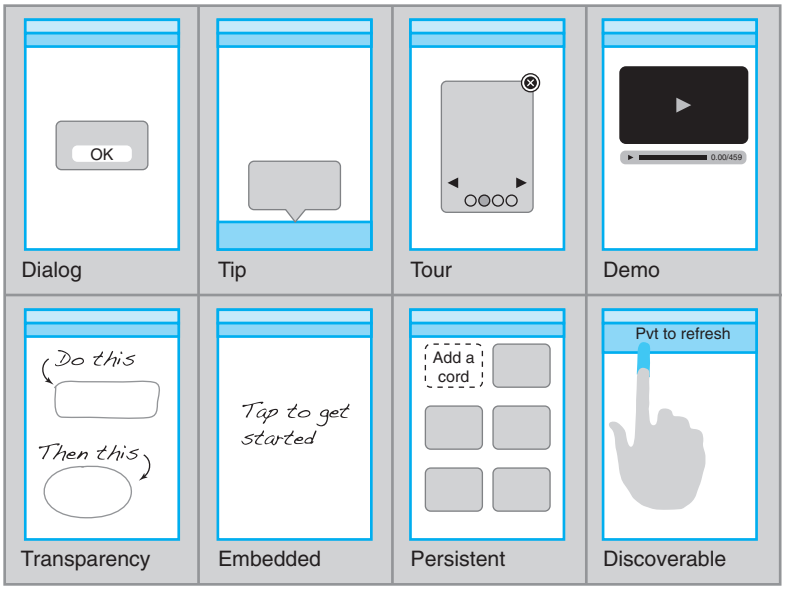
Check-in screens. How do I check in from a specific location, make a comment, and share comments with friends and followers on a social network?

Maps. How do I display a map within the context of an app that addresses some other subject? For example, review a restaurant and represent its location within a city.

Popovers. How do I represent a message or information (from the app or another user) that arises in real time or as the consequence of a user action?

FIGURE 16.3

Examples of mobile invitation patterns [Nei11]



Sign-up Flows. How do I provide a simple way to sign in or register for information or functionality?

Custom Tab Navigation. How do I represent a variety of different content objects in a manner that enables the user to select the one she wants?

Invitations. How do I inform the user that he must participate in some action or dialog? Typical examples are illustrated in Figure 16.3.

Additional information about mobile UI patterns can be found in [Nei12] and [Hoo12]. In addition to UI patterns, Meier and his colleagues [Mei12] propose a variety of more general pattern descriptions for mobile apps. Further information on mobile patterns, including an extensive pattern library[8] has been developed by Nokia [Nok13].

16.8 SUMMARY

Design patterns provide a codified mechanism for describing problems and their solution in a way that allows the software engineering community to capture design knowledge for reuse. A pattern describes a problem, indicates the context enabling the user to understand the environment in which the problem resides, and lists a system of forces that indicate how the problem can be interpreted

8 Pointers to a variety of mobile pattern libraries can also be found at: **http://4ourth.com/wiki/Other%20Mobile%20Pattern%20Libraries**

within its context and how the solution can be applied. In software engineering work, we identify and document generative patterns. These patterns describe an important and repeatable aspect of a system and then provide us with a way to build that aspect within a system of forces that is unique to a given context.

Architectural patterns describe broad-based design problems that are solved using a structural approach. Data patterns describe recurring data-oriented problems and the data modeling solutions that can be used to solve them. Component patterns (also referred to as design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture. Interface design patterns describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end users. WebApp patterns address a problem set that is encountered when building WebApps and often incorporates many of the other patterns categories just mentioned. Mobile patterns address the unique nature of the mobile interface and functionality and control elements that are specific to mobile platforms.

A framework provides an infrastructure in which patterns may reside and idioms describe programming language–specific implementation detail for all or part of a specific algorithm or data structure. A standard form or template is used for pattern descriptions. A pattern language encompasses a collection of patterns, each described using a standardized template and interrelated to show how these patterns collaborate to solve problems across an application domain.

Pattern-based design is used in conjunction with architectural, component-level, and user interface design methods. The design approach begins with an examination of the requirements model to isolate problems, define context, and describe the system of forces. Next, pattern languages for the problem domain are searched to determine if patterns exist for the problems that have been isolated. Once appropriate patterns have been found, they are used as a design guide.

Problems and Points to Ponder

16.1. Discuss the three "parts" of a design pattern and provide a concrete example of each from some field other than software.

16.2. What is the difference between a nongenerative and a generative pattern?

16.3. How do architectural patterns differ from component patterns?

16.4. What is a framework and how does it differ from a pattern? What is an idiom and how does it differ from a pattern?

16.5. Using the design pattern template presented in Section 16.1.3, develop a complete pattern description for a pattern suggested by your instructor.

16.6. Develop a skeletal pattern language for a sport with which you are familiar. You can begin by addressing the context, the system of forces, and the broad problems that a coach

and team must solve. You need only specify pattern names and provide a one-sentence description for each pattern.

16.7. Find five patterns repositories and present an abbreviated description of the types of patterns contained in each.

16.8. When Christopher Alexander says "good design cannot be achieved simply by adding together performing parts," what do you think he means?

16.9. Using the pattern-based design tasks noted in Section 16.2.3, develop a skeletal design for the "interior design system" described in Section 15.3.2.

16.10. Build a pattern-organizing table for the patterns you used in Problem 16.9.

16.11. Using the design pattern template presented in Section 16.1.3, develop a complete pattern description for the **Kitchen** pattern mentioned in Section 16.3.

16.12. The gang of four [Gam95] have proposed a variety of component patterns that are applicable to object-oriented systems. Select one (these are available on the Web) and discuss it.

16.13. Find three patterns repositories for user-interface patterns. Select one pattern from each and present an abbreviated description of it.

16.14. Find three patterns repositories for WebApp patterns. Select one pattern from each and present an abbreviated description of it.

16.15. Find three patterns repositories for mobile patterns. Select one pattern from each and present an abbreviated description of it.

Further Readings and Information Sources

Many books on pattern-based design have been written for software engineers. Gamma and his colleagues [Gam95] have written the seminal book on the subject. More recent contributions include books by Burris (*Programming in the Large with Design Patterns,* Pretty Print Press, 2012), Smith (*Elemental Design Patterns,* Addison-Wesley, 2012). Lasater (*Design Patterns,* Wordware Publishing, 2007), Holzner (*Design Patterns for Dummies,* For Dummies, 2006), Freeman and her colleagues (*Head First Design Patterns,* O'Reilly Media, 2005), and Shalloway and Trott (*Design Patterns Explained,* 2nd. ed., Addison-Wesley, 2004). Kent Beck (*Implementation Patterns,* Addison-Wesley, 2008) addresses patterns for coding and implementation issues that are encountered during the construction activity.

Other books focus on design patterns as they are supplied in specific application development and language environments. Contributions in this area include: Bowers and his colleagues (*Pro HTML5 and CSS3 Design Patterns,* Apress, 2011), Scott and Neil (*Designing Web Interfaces: Principles and Patterns for Rich Interactions*, O'Reilly, 2009), Tropashko and Burleson (*SQL Design Patterns: Expert Guide to SQL Programming,* Rampant Techpress, 2007), Mahemoff (*Ajax Design Patterns,* O'Reilly Media, 2006), Bevis (*Java Design Pattern Essentials,* Ability First Limited, 2010), Metsker and Wake (*Design Patterns in Java,* Addison-Wesley, 2006), Millett (*Professional ASP.NET Design Patterns,* Wrox, 2010), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples in C# and .NET,* Addison-Wesley, 2006), Stefanov (*JavaScript Patterns,* O'Reilly, 2010), Sweat (*PHP | Architect's Guide to PHP Design Patterns,* Marco Tabini & Associates, 2005), Paul (*Design Patterns in C#*), Metsker (*Design Patterns C#,* Addison-Wesley, 2004), Grand and Merrill (*Visual Basic .NET Design Patterns,* Wiley, 2003), Crawford and Kaplan (*J2EE Design Patterns,* O'Reilly Media, 2003), Juric et al. (*J2EE Design Patterns Applied,* Wrox Press, 2002), and Marinescu and Roman (*EJB Design Patterns,* Wiley, 2002),

Still other books address specific application domains. These include contributions by Kuchana (*Software Architecture Design Patterns in Java,* Auerbach, 2004), Joshi (*C++ Design*

Patterns and Derivatives Pricing, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns,* Addison-Wesley, 2002), and Schmidt and Rising (*Design Patterns in Communication Software,* Cambridge University Press, 2001).

Classic books by the architect Christopher Alexander (*Notes on the Synthesis of Form,* Harvard University Press, 1964, and *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, 1977) are worthwhile reading for a software designer who intends to fully understand design patterns.

A wide variety of information sources on pattern-based design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to pattern-based design can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

WebApp Design

17

Key Concepts

aesthetic design 377
architecture design 381
component-level design 387
content architecture 381
content design 379
content objects 379
design goals 374
design pyramid. . . 375

In his authoritative book on Web design, Jakob Nielsen [Nie00] states: "There are essentially two basic approaches to design: the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer." During the first decade of Web development, the artistic idea was the approach that many developers chose. Design occurred in an ad hoc manner and was usually conducted as HTML was generated. Design evolved out of an artistic vision that evolved as WebApp construction occurred.

Even today, many Web developers use WebApps as poster children for "limited design." They argue that WebApp immediacy and volatility mitigate against formal design; that design evolves as an application is built (coded), and that relatively little time should be spent on creating a detailed design model. This argument has merit, but only for relatively simple WebApps. When content and function are complex; when the size of the WebApp encompasses

Quick Look

What is it? Design for WebApps encompasses technical and nontechnical activities that include: establishing the look and feel of the WebApp, creating the aesthetic layout of the user interface, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the WebApp.

Who does it? Web engineers, graphic designers, content developers, and other stakeholders all participate in the creation of a WebApp design model.

Why is it important? Design allows you to create a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end users become involved in large numbers. Design is the place where WebApp quality is established.

What are the steps? WebApp design encompasses six major steps that are driven by information obtained during requirements modeling. Content design uses the content model (developed during analysis) as the basis for establishing the design of content objects. Aesthetic design (also called graphic design) establishes the look and feel that the end user sees. Architectural design focuses on the overall hypermedia structure of all content objects and functions. Interface design establishes the layout and interaction mechanisms that define the user interface. Navigation design defines how the end user navigates through the hypermedia structure, and component design represents the detailed internal structure of functional elements of the WebApp.

What is the work product? A design model that encompasses content, aesthetics, architecture, interface, navigation, and component-level design issues is the primary work product that is produced during WebApp design.

How do I ensure that I've done it right? Each element of the design model is reviewed in an effort to uncover errors, inconsistencies, or omissions. In addition, alternative solutions are considered, and the degree to which the current design model will lead to an effective implementation is also assessed.

design quality 372
graphic design 378
model-view-controller . . 384
navigation design 385
quality checklist. . . 374
webApp architecture 384

hundreds or thousands of content objects, functions, and analysis classes; and when the success of the WebApp will have a direct impact on the success of the business, design cannot and should not be taken lightly.

This reality leads us to Nielsen's second approach—"the engineering ideal of solving a problem for a customer." Web engineering[1] adopts this philosophy, and a more rigorous approach to WebApp design enables developers to achieve it.

17.1 WebApp Design Quality

Every person who has surfed the Web has an opinion about what makes a "good" WebApp. Individual viewpoints vary widely. Some users enjoy flashy graphics; others want simple text. Some demand copious information; others desire an abbreviated presentation. Some like sophisticated analytical tools or database access; others like to keep it simple. In fact, the user's perception of "goodness" (and the resultant acceptance or rejection of the WebApp as a consequence) might be more important than any technical discussion of WebApp quality.

But how is WebApp quality perceived? What attributes must be exhibited to achieve goodness in the eyes of end users and at the same time exhibit the technical characteristics of quality that will enable you to correct, adapt, enhance, and support the WebApp over the long term?

In reality, all of the technical characteristics of design quality discussed in Chapter 12 and the generic quality attributes presented in Chapter 19 apply to WebApps. However, the most relevant of these generic attributes—usability, functionality, reliability, efficiency, and maintainability—provide a useful basis for assessing the quality of Web-based systems.

Olsina and his colleagues [Ols99] have prepared a "quality requirement tree" that identifies a set of technical attributes—usability, functionality, reliability, efficiency, and maintainability—that lead to high-quality WebApps.[2] Figure 17.1 summarizes their work. The criteria noted in the figure are of particular interest if you design, build, and maintain WebApps over the long term.

? What are the major attributes of quality for WebApps?

Offutt [Off02] extends the five major quality attributes noted in Figure 17.1 by adding the following attributes:

> **Security.** WebApps have become heavily integrated with critical corporate and government databases. E-commerce applications extract

1 *Web engineering* [Pre08] is an adapted version of the software engineering approach that is presented throughout this book. It proposes an agile, yet disciplined framework for building industry-quality Web-based systems and applications.

2 These quality attributes are quite similar to those presented in Chapters 12 and 19. The implication: quality characteristics are universal for all software.

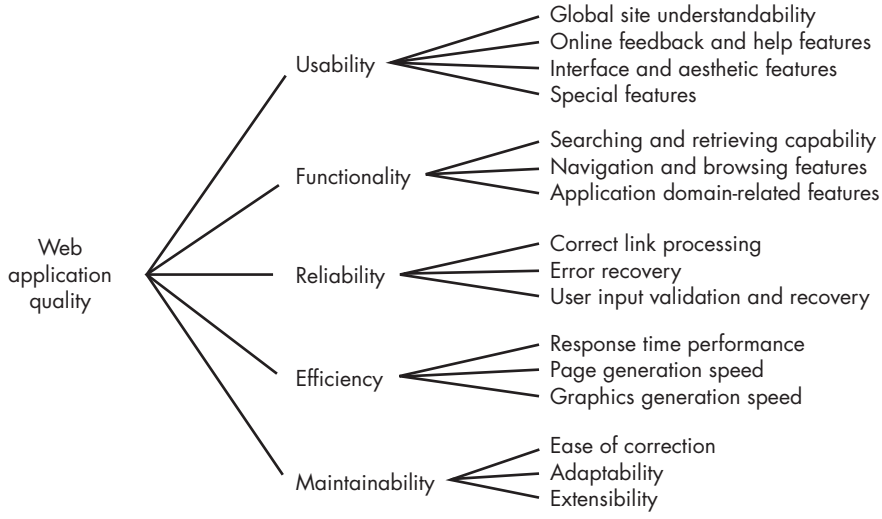


FIGURE 17.1
Quality requirements tree.
Source: [Ols99].

and then store sensitive customer information. For these and many other reasons, WebApp security is paramount in many situations. The key measure of security is the ability of the WebApp and its server environment to rebuff unauthorized access and/or thwart an outright malevolent attack. Security engineering is discussed in Chapter 27. For additional information on WebApp see [Web13], [Pri10], [Vac06], and [Kiz05].

Availability. Even the best WebApp will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a WebApp is available for use. But Offutt [Off02] suggests that "using features available on only one browser or one platform" makes the WebApp unavailable to those with a different browser/platform configuration. The user will invariably go elsewhere.

Scalability. Can the WebApp and its server environment be scaled to handle 100, 1,000, 10,000, or 100,000 users? Will the WebApp and the systems with which it is interfaced handle significant variation in volume or will responsiveness drop dramatically (or cease altogether)? It is important to design a WebApp that can accommodate the burden of success (i.e., significantly more end users) and become even more successful.

Time-to-Market. Although time-to-market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first WebApp to address a specific market segment often captures a disproportionate number of end users.

INFO

WebApp Design—Quality Checklist

The following checklist, adapted from information presented at **Webreference.com**, provides a set of questions that will help both Web engineers and end users assess overall WebApp quality:

- Can content and/or function and/or navigation options be tailored to the user's preferences?
- Can content and/or functionality be customized to the bandwidth at which the user communicates?
- Have graphics and other nontext media been used appropriately? Are graphics file sizes optimized for display efficiency?
- Are tables organized and sized in a manner that makes them understandable and displayed efficiently?
- Is HTML optimized to eliminate inefficiencies?
- Is the overall page design easy to read and navigate?
- Do all pointers provide links to information that is of interest to users?
- Is it likely that most links have persistence on the Web?
- Is the WebApp instrumented with site management utilities that include tools for usage tracking, link testing, local searching, and security?

Billions of Web pages are available for those in search of information. Even well-targeted Web searches result in an avalanche of content. With so many sources of information to choose from, how does the user assess the quality (e.g., veracity, accuracy, completeness, timeliness) of the content that is presented within a WebApp?

? What should we consider when assessing content quality?

Tillman [Til00] suggests a useful set of criteria for assessing the quality of content: Can the scope and depth of content be easily determined to ensure that it meets the user's needs? Can the background and authority of the content's authors be easily identified? Is it possible to determine the currency of the content, the last update, and what was updated? Are the content and its location stable (i.e., will they remain at the referenced URL)? Is content credible? Is content unique? That is, does the WebApp provide some unique benefit to those who use it? Is content valuable to the targeted user community? Is content well organized? Indexed? Easily accessible? These questions represent only a small sampling of the issues that should be addressed as the design of a WebApp evolves.

17.2 Design Goals

In her regular column on Web design, Jean Kaiser [Kai02] suggests a set of design goals that are applicable to virtually every WebApp regardless of application domain, size, or complexity:

Quote:

"Just because you can doesn't mean you should."

Jean Kaiser

Simplicity. Although it may seem old-fashioned, the aphorism "all things in moderation" applies to WebApps. There is a tendency among some designers to provide the end user with "too much"—exhaustive content, extreme visuals, intrusive animation, enormous Web pages, complex navigation, the list is long. Better to strive for moderation and simplicity.

Consistency. This design goal applies to virtually every element of the design model. Content should be constructed consistently (e.g., text formatting and

font styles should be the same across all text documents; graphic art should have a consistent look, color scheme, and style). Graphic design (aesthetics) should present a consistent look across all parts of the WebApp. Navigation mechanisms should be used consistently across all WebApp elements. As Kaiser [Kai02] notes: "Remember that to a visitor, a website is a physical place. It is confusing if pages within a site are not consistent in design."

Identity. The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built. A website for a music company will undoubtedly have a different look and feel than a WebApp designed for a financial services company.

Robustness. Based on the identity that has been established, a WebApp often makes an implicit "promise" to a user. The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.

Navigability. Navigation should be designed in a manner that is intuitive and predictable. That is, the user should understand how to move about the WebApp without having to search for navigation links or instructions. For example, if a field of graphic icons or images contains selected icons or images that will be used as navigation mechanisms, these must be identified visually. Nothing is more frustrating than trying to find the appropriate live link among many graphical images.

Visual Appeal. Of all software categories, Web applications are unquestionably the most visual, the most dynamic, and the most unapologetically aesthetic. Beauty (visual appeal) is undoubtedly in the eye of the beholder, but many design characteristics (e.g., the look and feel of content; interface layout; color coordination; the balance of text, graphics, and other media; navigation mechanisms) do contribute to visual appeal.

Compatibility. A WebApp will be used in a variety of environments (e.g., different hardware, Internet connection types, operating systems, browsers) and must be designed to be compatible with each.

Quote:

"To some, Web design focuses on visual look and feel . . . To others, Web design is about structuring information and navigation through the document space. Others might even consider Web design to be about the technology used to build interactive Web applications. In reality, design includes all of these things and maybe more."

Thomas Powell

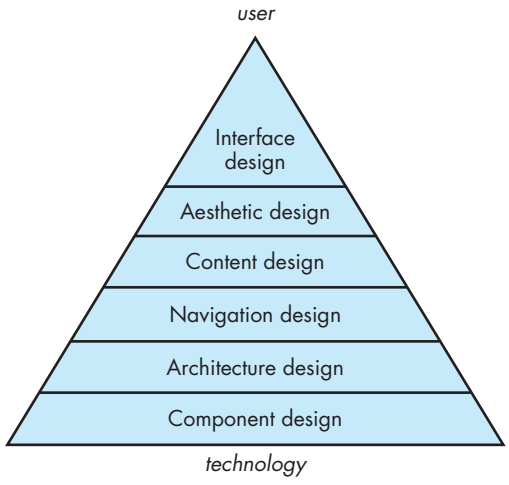
17.3 A Design Pyramid for WebApps

What is design in the context of Web engineering? This simple question is more difficult to answer than one might believe. Pressman and Lowe [Pre08] discuss this when they write:

> The creation of an effective design will typically require a diverse set of skills. Sometimes, for small projects, a single developer may need to be multi-skilled. For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists: Web engineers, graphic designers, content developers, programmers, database specialists, information architects, network engineers, security experts, and testers.

FIGURE 17.2

A design pyramid for WebApps



Quote:

"If a site is perfectly usable but it lacks an elegant and appropriate design style, it will fail."

Curt Cloninger

Drawing on these diverse skills allows the creation of a model that can be assessed for quality and improved *before* content and code are generated, tests are conducted, and end-users become involved in large numbers. If analysis is where *WebApp quality is established,* then design is where the *quality is truly embedded.*

The appropriate mix of design skills will vary depending upon the nature of the WebApp. Figure 17.2 depicts a design pyramid for WebApps. Each level of the pyramid represents a design action that is described in the sections that follow.

17.4 WebApp Interface Design

When a user interacts with a computer-based system, a set of fundamental principles and overriding design guidelines apply. These have been discussed in Chapter 15.[3] Although WebApps present a few special user interface design challenges, the basic principles and guidelines are applicable.

One of the challenges of interface design for WebApps is the indeterminate nature of the user's entry point. That is, the user may enter the WebApp at a "home" location (e.g., the home page) or may be linked into some lower level of the WebApp architecture. In some cases, the WebApp can be designed in a way that reroutes the user to a home location, but if this is undesirable, the WebApp design must provide interface navigation features that accompany all content objects and are available regardless of how the user enters the system.

The objectives of a WebApp interface are to: (1) establish a consistent window into the content and functionality provided by the interface, (2) guide the user through a series of interactions with the WebApp, and (3) organize the navigation options and content available to the user. To achieve a consistent interface, you

3 Section 15.5 is dedicated to WebApp interface design. If you have not already done so, read it at this time.

ADVICE

Not every Web engineer (or software engineer) has artistic (aesthetic) talent. If you fall into this category, hire an experienced graphic designer for aesthetic design work.

should first use aesthetic design (Section 17.5) to establish a coherent "look." This encompasses many characteristics, but must emphasize the layout and form of navigation mechanisms. To guide user interaction, you may draw on an appropriate metaphor[4] that enables the user to gain an intuitive understanding of the interface. To implement navigation options, you can select *navigation menus* positioned consistently on Web pages, *graphic icons* represented in a manner that enable a user to recognize that the icon is a navigation element, and/or *graphic images* that provide a link to a content object or WebApp functionality. It is important to note that one or more of these navigation mechanisms should be provided at every level of the content hierarchy.

17.5 Aesthetic Design

? What interaction mechanisms are available to WebApp designers?

Aesthetic design, also called *graphic design,* is an artistic endeavor that complements the technical aspects of WebApp design. Without it, a WebApp may be functional, but unappealing. With it, a WebApp draws its users into a world that embraces them on a visceral, as well as an intellectual level.

But what is aesthetic? There is an old saying, "beauty exists in the eye of the beholder." This is particularly appropriate when aesthetic design for WebApps is considered. To perform effective aesthetic design, return to the user hierarchy developed as part of the requirements model (Chapter 8) and ask, "Who are the WebApp's users and what "look" do they desire?"

SafeHome

Graphic Design

The scene: Doug Miller's office after the first web interface prototype review.

The players: Doug Miller (SafeHome software engineering project manager) and Vinod Raman member of the *SafeHome* software engineering team.

The conversation:

Doug: What's your impression of new Web page design?

Vinod: I like it, but more importantly, our customers like it.

Doug: How much help did you get from the graphic designer we borrowed from marketing?

Vinod: A lot, actually. She has a great eye for page layout and suggested an awesome graphic theme for the pages. Much better than what we came up with on our own.

Doug: That's good. Any issues?

Vinod: We still have to create alternate pages to take accessibility issues into account for some of our visually impaired users. But we would have had to do that for any Web page design we had.

Doug: Do we need graphic design help on the alternative pages as well?

Vinod: Sure. The designer has a good understanding of usability and accessibility issues.

Doug: OK, I'll ask marketing if we can borrow her a little longer.

4 In this context, a *metaphor* is a representation (drawn from the user's real-world experience) that can be modeled within the context of the interface. A simple example might be a slider switch that is used to control the auditory volume of an .mpg file.

17.5.1 Layout Issues

Every Web page has a limited amount of "real estate" that can be used to support nonfunctional aesthetics, navigation features, informational content, and user-directed functionality. The development of this real estate is planned during aesthetic design.

Quote:

"We find that people quickly evaluate a site by visual design alone."

Stanford Guidelines for Web Credibility

Like all aesthetic issues, there are no absolute rules when screen layout is designed. However, a number of general layout guidelines are worth considering:

Don't be afraid of open space. It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and create visual chaos that is not pleasing to the eye.

Emphasize content. After all, that's the reason the user is there. Nielsen [Nie00] suggests that the typical Web page user should be 80 percent content with the remaining real estate dedicated to navigation and other features.

Organize layout elements from top left to bottom right. The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top left to bottom right.[5] If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.

Group navigation, content, and function geographically within the page. Humans look for patterns in virtually all things. If there are no discernible patterns within a Web page, user frustration is likely to increase (owing to unnecessary searching for needed information).

Don't extend your real estate with the scrolling bar. Although scrolling is often necessary, most studies indicate that users would prefer not to scroll. It is often better to reduce page content or to present necessary content on multiple pages.

Consider resolution and browser window size when designing layout. Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of available space [Nie00]. With the growing use of mobile devices with different screen sizes, this concept becomes increasingly important.

17.5.2 Graphic Design Issues

Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with layout (Section 17.5.1) and proceeds into a consideration of global color schemes; type fonts, sizes, and styles; the use of

5 There are exceptions that are cultural and language-based, but this rule holds for most users.

supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.

A comprehensive discussion of graphic design issues for WebApps is beyond the scope of this book. You can obtain design tips and guidelines from many websites that are dedicated to the subject (e.g., **www.graphic-design.com**, **www.webdesignfromscratch.com**, **www.wpdfd.com**) or from one or more print resources (e.g., [Bea11], [McN10], [Lup08], and [Roc06]).

INFO

Well-Designed Websites

Sometimes, the best way to understand good WebApp design is to look at a few examples. In his article "The Top Twenty Web Design Tips," Marcelle Toor (**www.graphic-design.com/Web/feature/tips.html**) suggests the following websites as examples of good graphic design:

http://www.marcelletoordesigns.com/—ILR Website Design is a cutting-edge resource for clients developing their websites and other Internet ventures.

www.workbook.com—this site showcases work by illustrators and designers

www.pbs.org/riverofsong—a television series for public TV and radio about American music

www.RKDINC.com—a design firm with online portfolio and good design tips

http://www.creativehotlist.com/—a good source for well-designed sites developed by ad agencies, graphics arts firms, and other communications specialists

www.btdnyc.com—a design firm headed by Beth Toudreau

17.6 CONTENT DESIGN

Quote:

"Good designers can create normalcy out of chaos; they can clearly communicate ideas through the organizing and manipulating of words and pictures."

Jeffery Veen

Content design focuses on two different design tasks, each addressed by individuals with different skill sets. First, you should develop a design representation for content objects and the mechanisms required to establish their relationship to one another. Second, the information within a specific content object is created. The latter task may be conducted by copywriters, graphic designers, and others who generate the content to be used within a WebApp.

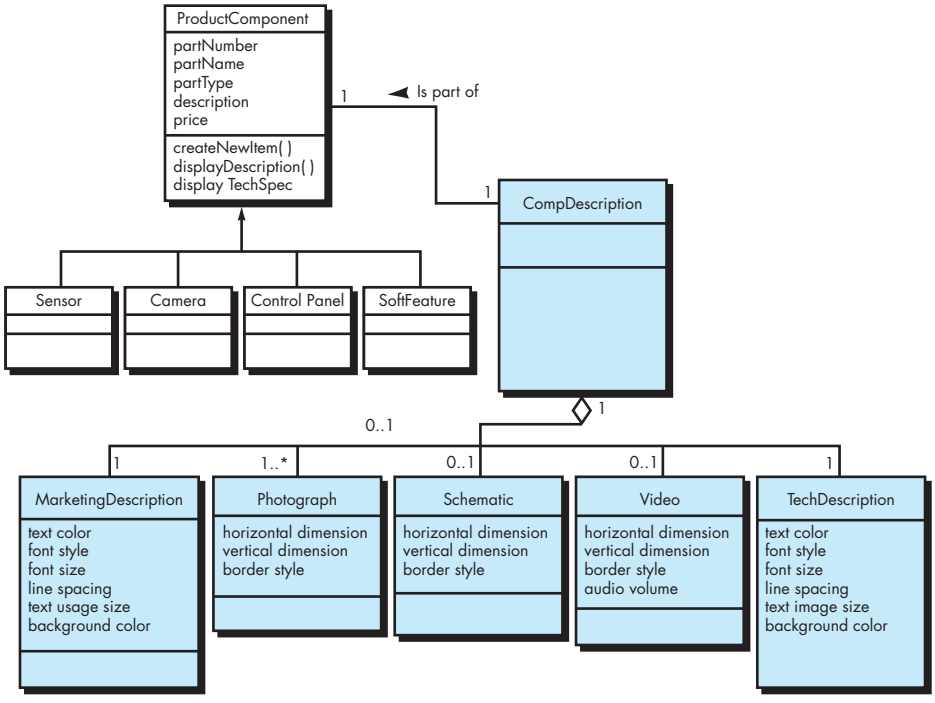
17.6.1 Content Objects

In the context of WebApp design, a content object is more closely aligned with a data object for traditional software. A *content object* has attributes that include content-specific information (normally defined during WebApp requirements modeling) and implementation-specific attributes that are specified as part of design.

As an example, consider an analysis class, **ProductComponent,** developed for the *SafeHome* e-commerce system**.** The analysis class attribute, **description**, is represented as a design class named **CompDescription** composed of five content objects: **MarketingDescription, Photograph, TechDescription, Schematic,** and **Video** shown as shaded objects noted in Figure 17.3. Information contained within the content object is noted as attributes. For example,

FIGURE 17.3

Design representation of content objects



Photograph (a .jpg image) has the attributes **horizontal dimension, vertical dimension,** and **border style.**

UML association and an aggregation[6] may be used to represent relationships between content objects. For example, the UML association shown in Figure 17.3 indicates that one **CompDescription** is used for each instance of the **ProductComponent** class. **CompDescription** is composed on the five content objects shown. However, the multiplicity notation shown indicates that **Schematic** and **Video** are optional (0 occurrences are possible), one **MarketingDescription** and one **TechDescription** are required, and one or more instances of **Photograph** are used.

17.6.2 Content Design Issues

Once all content objects are modeled, the information that each object is to deliver must be authored and then formatted to best meet the customer's needs. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) that will be used to deliver the information. Aesthetic design (Section 17.5) may also be applied to represent the proper look and feel for the content.

6 Both of these representations are discussed in Appendix 1.

ADVICE

Users tend to tolerate vertical scrolling more readily than horizontal scrolling. Avoid wide page formats.

As content objects are designed, they are "chunked" [Pow02] to form WebApp pages. The number of content objects incorporated into a single page is a function of user needs, constraints imposed by download speed of the Internet connection, and restrictions imposed by the amount of scrolling that the user will tolerate.

17.7 Architecture Design

Quote:

"[T]he architectural structure of a well designed site is not always apparent to the user—nor should it be."

Thomas Powell

Architecture design is tied to the goals established for a WebApp, the content to be presented, the users who will visit, and the navigation philosophy that has been established. As an architectural designer, you must identify content architecture and WebApp architecture. *Content architecture*[7] focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation. *WebApp architecture* addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design. Because the WebApp architecture may have a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design.

17.7.1 Content Architecture

The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from four different content structures [Pow02]:

? What types of content architectures are commonly encountered?

Linear structures (Figure 17.4) are encountered when a predictable sequence of interactions (with some variation or diversion) is common. A classic example might be a tutorial presentation in which pages of information along with related graphics, short videos, or audio are presented only after prerequisite information has been presented. The sequence of content presentation is predefined and generally linear. Another example might be a product order entry sequence in which specific information must be specified in a specific order. In such cases, the structures shown in Figure 17.4 are appropriate. As content and processing become more complex, the purely linear flow shown on the left of the figure gives way to more sophisticated linear structures in which alternative content may be invoked or a diversion to acquire complementary content (structure shown on the right side of Figure 17.4) occurs.

7 The term *information architecture* is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.

FIGURE 17.4

Linear structures

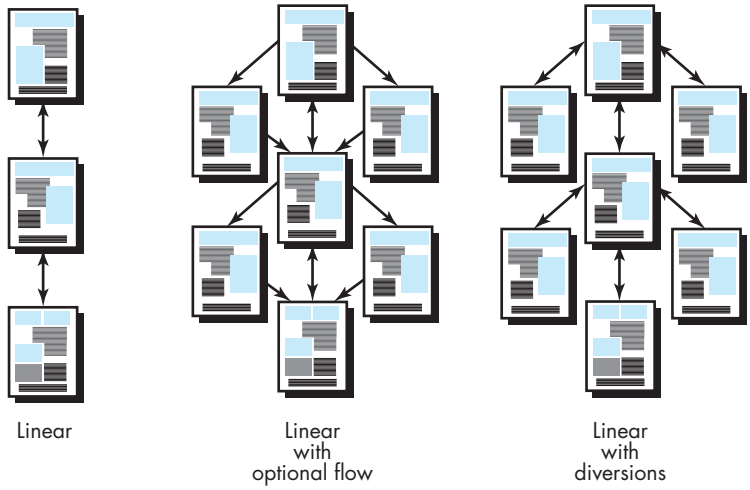
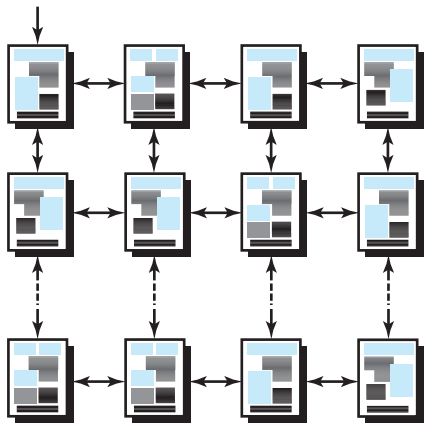


FIGURE 17.5

Grid structure



Grid structures (Figure 17.5) are an architectural option that you can apply when WebApp content can be organized categorically in two (or more) dimensions. For example, consider a situation in which an e-commerce site sells golf clubs. The horizontal dimension of the grid represents the type of club to be sold (e.g., woods, irons, wedges, putters). The vertical dimension represents the offerings provided by various golf club manufacturers. Hence, a user might navigate the grid horizontally to find the putters column and then vertically to examine the offerings provided by those manufacturers that sell putters. This WebApp architecture is useful only when highly regular content is encountered [Pow02].

Hierarchical structures (Figure 17.6) are undoubtedly the most common WebApp architecture. Unlike the partitioned software hierarchies discussed in Chapter 13 that encourage flow of control only along vertical branches of the

FIGURE 17.6

Hierarchical structure

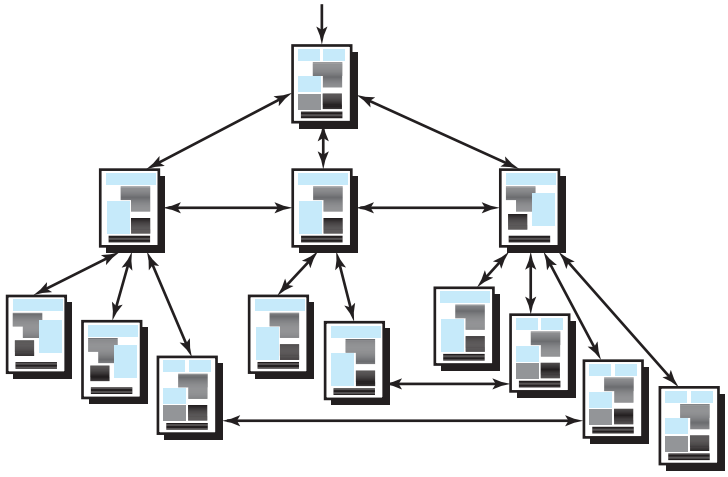
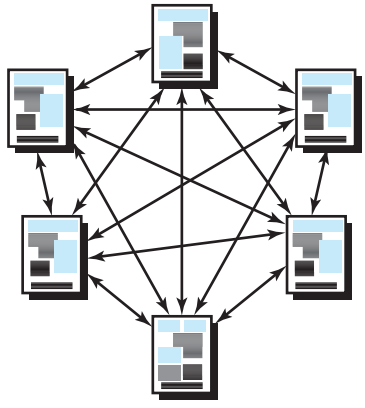


FIGURE 17.7

Network structure



hierarchy, a WebApp hierarchical structure can be designed in a manner that enables (via hypertext branching) flow of control horizontally, across vertical branches of the structure. Hence, content presented on the far left-hand branch of the hierarchy can have hypertext links that lead directly to content that exists in the middle or right-hand branch of the structure. It should be noted, however, that although such branching allows rapid navigation across WebApp content, it can lead to confusion on the part of the user.

A *networked* or *"pure web" structure* (Figure 17.7) is similar in many ways to the architecture that evolves for object-oriented systems. Architectural components (in this case, Web pages) are designed so that they may pass control (via hypertext links) to virtually every other component in the system. This approach allows considerable navigation flexibility, but at the same time, can be confusing to a user.

The architectural structures discussed in the preceding paragraphs can be combined to form *composite structures*. The overall architecture of a WebApp may be hierarchical, but part of the structure may exhibit linear characteristics, while another part of the architecture may be networked. Your goal as an architectural designer is to match the WebApp structure to the content to be presented and the processing to be conducted.

17.7.2 WebApp Architecture

WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives. Jacyntho and his colleagues [Jac02b] describe the basic characteristics of this infrastructure in the following manner:

> Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages).

The authors suggest a three-layer design architecture that decouples interface from navigation and from application behavior. They argue that keeping interface, application, and navigation separate simplifies implementation and enhances reuse.

KEY POINT
The MVC architecture decouples the user interface from WebApp functionality and information content.

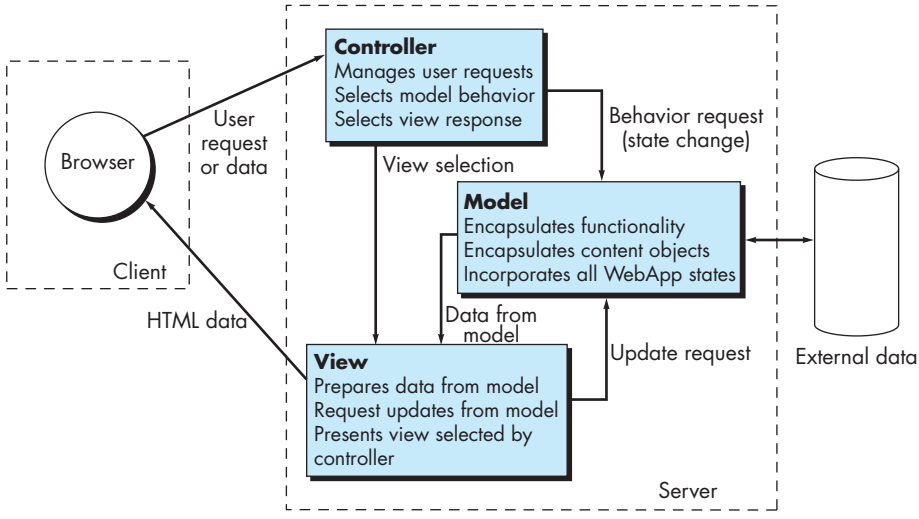
The *Model-View-Controller* (MVC) architecture [Kra88][8] is one of a number of suggested WebApp infrastructure models that decouple the user interface from the WebApp functionality and informational content. The *model* (sometimes referred to as the "model object") contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific. The *view* contains all interface-specific functions and enables the presentation of content and processing logic, including all content objects, access to external data/information sources, and all processing functionality required by the end user. The *controller* manages access to the model and the view and coordinates the flow of data between them. In a WebApp, "the view is updated by the controller with data from the model based on user input" [WMT02]. A schematic representation of the MVC architecture is shown in Figure 17.8.

Referring to the figure, user requests or data are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in

8 It should be noted that MVC is actually an architectural design pattern developed for the Smalltalk environment (see **www.smalltalk.org**) and can be used for any interactive application.

FIGURE 17.8

The MVC architecture
Source: Adapted from [Jac02b].



a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine.

In many cases, WebApp architecture is defined within the context of the development environment in which the application is to be implemented. If you have further interest, see [Fow03] for a discussion of development environments and their role in the design of Web application architectures.

17.8 NAVIGATION DESIGN

Quote:

"Just wait, Gretel, until the moon rises, and then we shall see the crumbs of bread which I have strewn about, they will show us our way home again."

Hansel and Gretel

Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, identify the semantics of navigation for different users of the site, and define the mechanics (syntax) of achieving the navigation.

17.8.1 Navigation Semantics

Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases (Chapter 9) developed for each category of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. In addition, the use cases developed for each actor will define a set of classes that encompass one or more

KEY POINT

An NSU describes the navigation requirements for each use case. In essence, the NSU shows how an actor moves between content objects or WebApp functions.

content objects or WebApp functions. As each user interacts with the WebApp, she encounters a series of *navigation semantic units* (NSUs)—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements" [Cac02].

An NSU is composed of a set of navigation elements called *ways of navigating* (WoN) [Gna99]. A WoN represents the best navigation pathway to achieve a navigational goal for a specific type of user. Each WoN is organized as a set of *navigational nodes* (NN) that are connected by navigational links. In some cases, a navigational link may be another NSU. Therefore, the overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

To illustrate the development of an NSU, consider the use case **Select SafeHome Components**:

Use Case: Select SafeHome Components

The WebApp will recommend product components (e.g., control panels, sensors, cameras) and other features (e.g., PC-based functionality implemented in software) for each room and exterior entrance. If I request alternatives, the WebApp will provide them, if they exist. I will be able to get descriptive and pricing information for each product component. The WebApp will create and display a bill-of-materials as I select various components. I'll be able to give the bill-of-materials a name and save it for future reference (see use case **Save Configuration**).

The underlined items in the use-case description represent classes and content objects that will be incorporated into one or more NSUs that will enable a new customer to perform the scenario described in the **Select SafeHome Components** use case.

Quote:

"The problem of website navigation is conceptual, technical, spatial, philosophical and logistic. Consequently, solutions tend to call for complex improvisational combinations of art, science and organizational psychology."

Tim Horgan

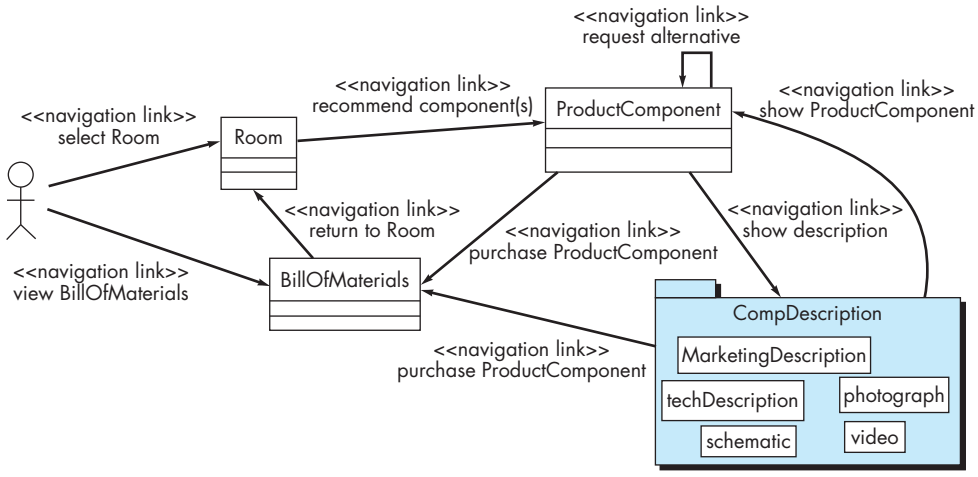
Figure 17.9 depicts a partial semantic analysis of the navigation implied by the **Select SafeHome Components** use case. Using the terminology introduced earlier, the figure also represents a way of navigating (WoN) for the **SafeHomeAssured.com** WebApp. Important problem domain classes are shown along with selected content objects (in this case the package of content objects named **CompDescription**, an attribute of the **ProductComponent** class). These items are navigation nodes. Each of the arrows represents a navigation link[9] and is labeled with the user-initiated action that causes the link to occur.

You can create an NSU for each use case associated with each user role. For example, a **new customer** for **SafeHomeAssured.com** may have three different use cases, all resulting in access to different information and WebApp functions. An NSU is created for each goal.

During the initial stages of navigation design, the WebApp content architecture is assessed to determine one or more WoN for each use case. As noted

9 These are sometimes referred to as *navigation semantic links* (NSL) [Cac02].

FIGURE 17.9 Creating an NSU



earlier, a WoN identifies navigation nodes (e.g., content) and then links that enable navigation between them. The WoN are then organized into NSUs.

ADVICE

In most situations, choose either horizontal or vertical navigation mechanisms, but not both.

17.8.2 Navigation Syntax

As design proceeds, your next task is to define the mechanics of navigation. Most websites make use of one or more of the following navigation options for implementing each NSU: individual navigation links, horizontal or vertical navigation bars (lists), tabs, or access to a complete site map.

ADVICE

The site map should be accessible from every page. The map itself should be organized so that the structure of WebApp information is readily apparent.

In addition to choosing the mechanics of navigation, you should also establish appropriate navigation conventions and aids. For example, icons and graphical links should look "clickable" by beveling the edges to give the image a three-dimensional look. Audio or visual feedback should be designed to provide the user with an indication that a navigation option has been chosen. For text-based navigation, color should be used to indicate navigation links and to provide an indication of links already traveled. These are but a few of dozens of design conventions that make navigation user-friendly.

17.9 COMPONENT-LEVEL DESIGN

Modern WebApps deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that are appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, and (4) establish data interfaces with external corporate systems. To achieve these (and many other) capabilities, you must design and

construct program components that are identical in form to software components for traditional software.

The design methods discussed in Chapter 14 apply to WebApp components with little, if any, modification. The implementation environment, programming languages, and design patterns, frameworks, and software may vary somewhat, but the overall design approach remains the same.

17.10 Summary

The quality of a WebApp—defined in terms of usability, functionality, reliability, efficiency, maintainability, security, scalability, and time-to-market—is introduced during design. To achieve these quality attributes, a good WebApp design should exhibit the following characteristics: simplicity, consistency, identity, robustness, navigability, and visual appeal. WebApp design activity focuses on six different elements of the design.

Interface design describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. A set of interface design principles and an interface design workflow guide you when layout and interface control mechanisms are designed.

Aesthetic design, also called graphic design, describes the "look and feel" of the WebApp and includes color schemes; geometric layout; text size, font, and placement; the use of graphics; and related aesthetic decisions. A set of graphic design guidelines provides the basis for a design approach.

Content design defines the layout, structure, and outline for all content that is presented as part of the WebApp and establishes the relationships between content objects. Content design begins with the representation of content objects, their associations, and relationships. A set of browsing primitives establishes the basis for navigation design.

Architecture design identifies the overall hypermedia structure for the WebApp and encompasses both content architecture and WebApp architecture. Architectural styles for content include linear, grid, hierarchical, and network structures. WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives.

Navigation design represents the navigational flow between content objects and for all WebApp functions. Navigation semantics are defined by describing a set of navigation semantic units. Each unit is composed of ways of navigations and navigational links and nodes. Navigation syntax depicts the mechanisms used for effecting the navigation described as part of the semantics.

Component design develops the detailed processing logic required to implement functional components that implement a complete WebApp function.

Design techniques described in Chapter 14 are applicable for the engineering of WebApp components.

Problems and Points to Ponder

17.1. Why is the "artistic ideal" an insufficient design philosophy when modern WebApps are built? Is there ever a case in which the artistic ideal is the philosophy to follow?

17.2. In this chapter we select a broad array of quality attributes for WebApps. Select the three that you believe are most important, and make an argument that explains why each should be emphasized in WebApp design work.

17.3. Add at least five additional questions to the WebApp Design—Quality Checklist presented in Section 17.1.

17.4. You are a WebApp designer for FutureLearning Corporation, a distance learning company. You intend to implement an Internet-based "learning engine" that will enable you to deliver course content to a student. The learning engine provides the basic infrastructure for delivering learning content on any subject (content designers will prepare appropriate content). Develop a prototype interface design for the learning engine.

17.5. What is the most aesthetically pleasing website you have ever visited and why?

17.6. Consider the content object **Order,** generated once a user of **SafeHomeAssured.com** has completed the selection of all components and is ready to finalize his purchase. Develop a UML description for **Order** along with all appropriate design representations.

17.7. What is the difference between content architecture and WebApp architecture?

17.8. Reconsidering the FutureLearning "learning engine" described in Problem 17.4, select a content architecture that would be appropriate for the WebApp. Discuss why you made the choice.

17.9. Use UML to develop three or four design representations for content objects that would be encountered while the "learning engine" described in Problem 17.4 is designed.

17.10. Do a bit of additional research on the MVC architecture and decide whether it would be an appropriate WebApp architecture for the "learning engine" discussed in Problem 17.4.

17.11. What is the difference between navigation syntax and navigation semantics?

17.12. Define two or three NSUs for the **SafeHomeAssured.com** WebApp. Describe each in some detail.

Further Readings and Information Sources

Van Duyne and his colleagues (*The Design of Sites,* 2nd ed., Prentice Hall, 2007) have written a comprehensive book that covers most important aspects of the WebApp design process. Design process models and design patterns are covered in detail. Johnson (*Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules,* Morgan Kaufman, 2010) has written a book on user interaction design with many examples applicable to web design. Wodtke and Gavella (*Information Architecture: Blueprints for the Web,* 2nd ed., New Riders Publishing, 2009), Rosenfeld and Morville (*Information Architecture for the World Wide Web,* 3rd ed., O'Reilly & Associates, 2006), and Reiss (*Practical Information Architecture,* Addison-Wesley, 2000) address content architecture and other topics.

Although hundreds of books have been written on "Web design," very few of these discuss any meaningful technical methods for doing design work. At best, a variety of useful

guidelines for WebApp design is presented, worthwhile examples of Web pages and Java programming are shown, and the technical details important for implementing modern WebApps are discussed. Among the many offerings in this category are books by Butler (*The Strategic Web Designer,* How Books, 2013), Campos (*Web Design Source Book,* PromoPress, 2013), DeFederici (*The Web Designer's Roadmap,* Sitepoint, 2012), Robbins (*Learning Web Design,* O'Reilly Media, 2012), Sklar (*Principles of Web Design,* 5th ed., Course Technology, 2011), Cederholm (*Bulletproof Web Design*, 3rd ed., New Riders Press, 2011), and Shelly and his colleagues (*Web Design,* 4th ed., Course Technology, 2011). Offerings by Zeldman and Marcotte (*Designing with Web Standards,* 3rd ed., New Riders Publishing, 2009), McIntire (*Visual Design for the Modern Web,* New Riders Press, 2007), Watrall and Siarto *(Head First Web Design*, O-Reilly, 2008), Niederst *(Web Design in a Nutshell,* 3rd ed., O-Reilly, 2006), and Eccher (*Professional Web Design,* Course Technology, 2010) and (*Advanced Professional Web Design,* Charles River Media, 2006) are also worthy of consideration.

Books by Beaird (*The Principles of Beautiful Web Design,* 2nd ed., SitePoint, 2010), Clarke and Holzschlag (*Transcending CSS: The Fine Art of Web Design,* New Riders Press, 2006), and Golbeck (*Art Theory for Web Design,* Addison-Wesley, 2005) emphasize aesthetic design and are worthwhile reading for practitioners who have little background in the subject.

The agile view of design (and other topics) for WebApps is presented by Wallace and his colleagues (*Extreme Programming for Web Projects,* Addison-Wesley, 2003). Conallen (*Building Web Applications with UML,* 2nd ed., Addison-Wesley, 2002) and Rosenberg and Scott (*Applying Use-Case Driven Object Modeling with UML,* Addison-Wesley, 2001) present detailed examples of WebApps modeled using UML.

A wide variety of information sources on design for WebApps is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

MobileApp Design

18

Key Concepts

challenges. 392
cloud computing . . .405
context-aware apps. 399
design
best practices. . 401
goals. 396
mistakes 401

Mobile devices—smartphones, tablets, wearable devices, and other specialized products—have become the next wave of computing. In August 2012, *The Los Angeles Times* [Rod12] reported:

> For the first time ever more Americans own smartphones than regular phones or feature phones, a new report says.
>
> The report, put out by Chetan Sharma Consulting[1], shows smartphone penetration has crossed the 50% mark for the first time in the U.S.

Quick Look

What is it? MobileApp design encompasses technical and nontechnical activities that include: establishing the look and feel of the mobile application, creating the aesthetic layout of the user interface, establishing the rhythm of user interaction, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the MobileApp. Special attention needs to be given to the elements that add context awareness to the MobileApp.

Who does it? Software engineers, graphic designers, content developers, security specialists, and other stakeholders all participate in the creation of a MobileApp design model.

Why is it important? Design allows you to create a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end users become involved in large numbers. Design is the place where MobileApp quality is established.

What are the steps? MobileApp design is similar to WebApp design and encompasses six major steps that are driven by information obtained during requirements modeling. Content design addresses the same issues for both WebApp and MobileApp design. During architectural design, MobileApp developers determine which functions will be implemented in the native app running on the mobile device and which will be implemented as Web or cloud services. Interface design establishes the layout and interaction mechanisms that define the user experience. Ensuring that the MobileApp makes appropriate use of context affects both interface design and content design. Navigation design defines how the end user navigates through the content structure, and component design represents the detailed internal structure of functional elements of the MobileApp.

What is the work product? A design model that encompasses content, aesthetics, architecture, interface, navigation, and component-level design issues is the primary work product that is produced during MobileApp design.

How do I ensure that I've done it right? Each element of the design model is reviewed in an effort to uncover errors, inconsistencies, or omissions. In addition, alternative solutions are considered, and the degree to which the current design model will lead to an effective implementation is also assessed.

1 See **http://www.chetansharma.com/USmarketupdateQ22012.htm**

developing MobileApps 395
mobility environments 403
quality checklist .. 397
technical considerations.... 393
user interface design 398

And the trend is not exclusive to the United State's top four carriers. As GigaOM[2] points out, regional operators and other small carriers are also "tapping into the trove of cheaper smartphones in the market."

The GartnerGroup [Gar12] reports that in that same business quarter, 419 million smartphones were sold globally and projected annual sales of 119 million tablet computers, nearly a 100 percent increase over the previous year. Mobile computing has become a dominant force.

18.1 The Challenges

While mobile devices have many features in common with each other, their users often have very different perceptions of what features they expect to be bundled in each. Some users expect the same features that are provided on their personal computers. Others focus on the freedom that portable devices give them and gladly accept the reduced functionality in the mobile version of a familiar software product. Still others expect unique experiences not possible on traditional computing or entertainment devices. The user's perception of "goodness" might be more important than any of the technical quality dimensions of the MobileApp itself.

18.1.1 Development Considerations

Like all computing devices, mobile platforms are differentiated by the software they deliver—a combination of operating system (e.g., Android or iOS) and a small subset of the hundreds of thousands of MobileApps that provide a very wide range of functionality. New tools allow individuals with little formal training to create and sell apps alongside other apps developed by large teams of software developers.

? What makes MobileApp development different and challenging?

Even though apps can be developed by amateurs, many software engineers think that MobileApps are among the most challenging software systems being built today [Voa12]. Mobile platforms are very complex. Both the Android and iOS operating systems contain over 12 million lines of code. Mobile devices often have mini browsers that will not display the full set of content available on a Web page. Different mobile devices use different operating systems and platform dependent development environments. Mobile devices tend to have smaller and more varied screen sizes than personal computers. This may require greater attention to user interface design issues, including decisions to limit display of some content. In addition, MobileApps must be designed to take into account intermittent connectivity outages, limitations on battery life, and other device constraints[3] [Whi08].

2 See **http://gigaom.com/mobile/carrier-data-confirms-it-half-of-us-now-owns-a-smartphone/**
3 Available at **http://www.devx.com/SpecialReports/Article/37693**

System components in mobile computing environments are likely to change their locations while MobileApps are running. In order to maintain connectivity in nomadic networks,[4] coordination mechanisms for discovering devices, exchanging information, maintaining security and communication integrity, and synchronizing actions must be developed.

KEY POINT

There is always a trade-off between security and other elements of the MobileApp design.

In addition, software engineers must identify the proper design trade-offs between the expressive power of the MobileApp and stakeholder security concerns. Developers must seek to discover algorithms (or adapt existing algorithms) that are energy efficient in order to conserve battery power when possible. Middleware may have to be created to allow different types of mobile devices to communicate with each other in the same mobile networks [Gru00].

Software engineers should craft a user experience that takes advantage of device characteristics and context-aware applications. The nonfunctional requirements (e.g., security, performance, usability) are a bit different from those for either WebApps or desktop software applications. Testing MobileApps (Chapter 26) provides additional challenges because the user expects that they will work in a large number of physically different environments. Because MobileApps often execute on a variety of device platforms, portability is a important consideration. In addition, the time and effort associated with accommodating multiple platforms often increases overall project cost [Was10].

18.1.2 Technical Considerations

The low cost of adding Web capabilities to everyday devices such as phones, cameras, and TVs is transforming the way people access information and use network services [Sch11]. Among the many technical considerations that MobileApps should address are the following:

? What are the primary technical considerations when building a MobileApp?

Multiple hardware and software platforms. It is not at all unusual for a MobileApp to run on many different platforms (both mobile and stationary) with a range of differing levels of functionality. The reasons for these differences are in part because the hardware and software available are quite different from device to device. This increases both development cost and time. It also can make configuration management (Chapter 29) more difficult.

Many development frameworks and programming languages. MobileApps are currently being written in at least three distinct programming languages (Java, Objective C, and C#) for at least five popular development frameworks (Android, iOS, BlackBerry, Windows, Symbian) [Was10]. Very few mobile devices allow direct development on a device itself. Instead, MobileApp developers use emulators running on desktop development systems. These

4 *Nomadic networks* have changing connections to mobile devices or servers.

emulators may or may not accurately reflect the limitations of the device itself. Thin-client applications are often easier to port to multiple devices than applications designed to run exclusively on the mobile device.

Many app stores with different rules and tools. Each mobile platform has its own app store and its own standards for accepting apps (e.g., Apple,[5] Google,[6] RIM,[7] Microsoft,[8] and Nokia[9] publish their own standards). Development of a MobileApp for multiple platforms must proceed separately, and each version of the MobileApp needs its own standards expert.

Very short development cycles. Software engineers often use of agile development processes when building MobileApps in an effort to reduce development time [Was10].

UI limitations and complexities of interaction with sensors and cameras. Mobile devices have smaller screen sizes than personal computers and a richer set of interaction possibilities (e.g., voice, touch, gesture, eye tracking) and usage scenarios based on context awareness. The style and appearance of the user interface is often dictated by the nature of platform-specific development tools [Rot02]. Allowing smart devices to interact with smart spaces offers the potential to create personalized, networked, high-fidelity application platforms such as those seen by merging smartphones and car infotainment systems.[10]

Effective use of context. Users expect MobileApps to deliver personalized user experiences based on the physical location of a device in relation to the available network features. User interface design and context-aware applications are discussed in greater detail in Section 18.2.

Power management. Battery life is often one of the most limiting constraints on MobileApps. Backlighting, reading and writing to memory, using wireless connections, making use of specialized hardware, and processor speed all impact power usage and need to be taken into account by software developers [Mei09].

Security and privacy models and policies. Wireless communication is difficult to protect from eavesdropping. Indeed preventing *man-in-the-middle-attacks*[11] in automotive applications can be critical to the safety of the users [Bos11]. Data stored on a mobile device is subject to theft if a device is lost or a malicious app is downloaded. Software policies that increase the level

5 **https://developer.apple.com/appstore/guidelines.html**

6 **http://developer.android.com/distribute/googleplay/publish/preparing.html**

7 **https://appworld.blackberry.com/isvportal/guidelines.do**

8 **http://msdn.microsoft.com/en-us/library/ff941089%28v=vs.92%29.aspx**

9 **http://support.publish.nokia.com/?p=64**

10 When used in an automotive setting, smart devices should be able to restrict access to services that may distract the driver and allow hands-free operation when a vehicle is moving [Bos11].

11 These attacks involve a third party intercepting communications between two trusted sources and impersonating one or both of the parties.

of confidence in the security and privacy of a MobileApp often reduce the usability of the app and the spontaneity of the communication among users [Rot02].

Computational and storage limitations. There is great interest in using mobile devices to control home environmental and security services. When MobileApps are allowed to interact with devices and services in their environment, it is easy to overwhelm the mobile device (storage, processing speed, power consumed) with the sheer volume of information [Spa11]. Developers may need to look for programming shortcuts and means of reducing the demands made on processor and memory resources.

Applications that depend on external services. Building thin mobile clients suggests the need to rely on Web service providers and cloud storage facilities. This increases concerns for both data or service accessibility and security [Rot02].

Testing complexity. Thin-client MobileApps are particularly challenging to test.[12] They exhibit many of the same testing challenges found in WebApps (Chapter 25), but they have the additional concerns associated with transmission of data through Internet gateways and telephone networks [Was10]. Testing of MobileApps will be discussed in Chapter 26.

18.2 Developing MobileApps

Andreu [And05] describes a spiral engineering process model for MobileApp design containing six activities:

KEY POINT

WebApp development makes use of an agile, spiral engineering process model.

Formulation. Involves architectural design, navigation design, the goals, features, and functions of the MobileApp are identified to determine the scope and the size of the first increment. Developers must be conscious of human, social, cultural, and organizational activities that may reveal hidden aspects of the users' needs and affect the business targets and functionality of the proposed MobileApp.

Planning. The total project costs and risks are determined. The detailed schedule is set and the process for the next increments is documented.

Analysis. All mobile user requirements are specified and the content items that will be needed are identified. Actions include content analysis, interaction analysis, functional analysis, and configuration analysis. It is at this stage that developers identify whether they will build a thin or fat client. Identifying the nature of the user goals (informational or transactional) will help to determine the type of MobileApp that needs to be developed.

12 MobileApps that run entirely on the device can be tested using traditional software testing methods (Chapter 23) or using emulators running on personal computers.

Engineering. Involves architectural design, navigation design, interface design, content design, and content production. Software engineers examine the constraints imposed by the targeted mobile devices, including considerations imposed by the wireless network technologies chosen and the nature of the Web services required to implement the MobileApp.

Implementation and Testing. During this activity, the MobileApp is coded and tested. Among the issues that can make testing a challenge are: (1) high loss rates due to radio interference and frequent disconnection due to network coverage issues, (2) frequent data transmission delays due to relatively low bandwidth, and (3) security concerns because mobile devices are less secure and relatively easy to attack.

User Evaluation. The MobileApp is assessed for usability and accessibility then the formulation process begins for the next increment.

? What should we consider when assessing MobileApp quality?

Andreou [And05] suggests that ubiquity, personalization, flexibility, and localization should be overriding design goals for every MobileApp. Mobile users expect to have the ability to receive information and conduct transactions in real time regardless of their physical location or the number of concurrent users. Mobile applications should present services and applications that are customized according to the preferences of the user. Users of mobile devices should be able to engage in activities such as receiving information or conducting transactions with ease. Mobile users should have access to local information and services. This implies recognizing the importance or context when designing the MobileApp user experience.

SafeHome

Formulating MobileApp Requirements

The scene: A meeting room. The first meeting to identify requirements for a mobile version of the *SafeHome* WebApp.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function present in the WebApp.

Vinod (interrupting): My understanding is that people want *SafeHome* functionality to be accessible from mobile devices as well . . . including the home security function?

Marketing person: Yes, that's right . . . we'll have to add that functionality and try to make it context aware to help personalize the user experience.

Facilitator: Context aware in what sense?

Marketing person: People might want to use a smartphone instead of the control panel and avoid logging on to a website when they are in the driveway at home. Or they might not want all family members to have access to the master control dashboard for the system from their phones.

Facilitator: Do you have specific mobile devices in mind?

Marketing person: Well, all smartphones would be nice. We will have a Web version done, so won't the MobileApp run on all of them?

Jamie: Not quite. If we took a mobile phone browser approach we might be able to reuse a lot of our WebApps. But remember, smartphone screen sizes vary and they may or may not all have the same touch capabilities. So at the very least we would have to create a mobile website that takes the device features into account.

Ed: Perhaps we should build the mobile version of the website first.

Marketing person: OK, but a mobile website solution wasn't what we had in mind.

Vinod: Each mobile platform seems to have its own unique development environment too.

Production rep: Can we restrict MobileApp development to only one or two types of smartphones?

Marketing person: I think that might work. Unless I'm mistaken, the smartphone market is dominated by two smartphone platforms right now.

Jamie: There's also security to worry about. We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Also a phone could get lost or stolen more easily than a laptop.

Doug: Very true.

Marketing: But we still need the same level of security . . . just also be sure to stop an outsider from getting in with a stolen phone.

Ed: That's easier said than done and . . .

Facilitator (interrupting): Let's not worry about those details yet.
(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: As a starting point, can we identify which elements of WebApp security function are needed in the MobileApp and which will need to be newly created? Then we can decide how many mobile platforms we can support and when we can move forward on this project.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

18.2.1 MobileApp Quality

In reality, almost every quality dimension and factor discussed in Chapter 19 applies to MobileApps. However, Andreou [And05] suggests that end-user satisfaction with a MobileApp is dictated by six important quality factors: functionality, reliability, usability, efficiency, maintainability, and portability.

INFO

MobileApp —Quality Checklist

The following checklist provides a set of questions that will help both software engineers and end users assess overall MobileApp quality:

- Can content and/or function and/or navigation options be tailored to the user's preferences?
- Can content and/or functionality be customized to the bandwidth at which the user communicates? Does the app account for weak or lost signals in an acceptable manner?
- Can content and/or function and/or navigation options be made context aware according to the user's preferences?
- Has adequate consideration been given to the power availability on the target device(s)?
- Have graphics, media (audio, video), and other Web or cloud services been used appropriately?
- Is the overall page design easy to read and navigate? Does the app take screen-size differences into account?
- Does the user interface conform to the display and interaction standards adopted for the targeted mobile device(s)?
- Does the app conform to the reliability, security, and privacy expectations of its users?
- What provisions have been made to ensure an app remains current?
- Has the MobileApp been tested in all targeted user environments and for all targeted devices?

18.2.2 User Interface Design

Mobile device users expect that minimal learning time will be required to master a MobileApp. To achieve this, MobileApp designers use consistent icon representations and placement across multiple platforms. In addition, designers must be sensitive to the user's expectation of privacy with regard to the display of personal information on the screen of the mobile device. Touch and gesture interfaces, along with sophisticated voice input, are maturing rapidly [Shu12] and have already become part of the user interface designer's toolbox.

KEY POINT

Accessibility is an important design issue and must be considered when user-centered design is applied.

Legal and ethical pressure to provide for access by all persons suggest that mobile device interfaces need to account for brand differences, cultural differences, differences in computing experience, elderly users, and users with disabilities (e.g., visual, aural, mobility). The effects of poor usability may mean that users cannot complete their tasks or will not be satisfied with the results. This suggests the importance of user-centered design activities in each of the usability areas (user interface, external accessory interface, and service interface). In trying to meet stakeholder usability expectations, MobileApp developers should attempt to answer these questions to assess the out-of-the-box readiness of the device:

- Is the user interface consistent across applications?
- Is the device interoperable with different network services?
- Is the device acceptable in terms of stakeholder values[13] in the target market area?

Eisenstein [Eis01] claims that the use of abstract, platform-neutral models to describe a user interface greatly facilitates the development of consistent, usable multiplatform user interfaces for mobile devices. Called *model based design,* this approach uses three different models. A *platform model* describes the constraints imposed by each platform to be supported. A *presentation model* describes the appearance of the user interface. The *task model* is a structured representation of the tasks a user needs to perform to meet her task goals. In the best case, model-based design (Chapter 12) involves the creation of databases that contain the models and has tool support for generating user interfaces for multiple devices automatically. Utilizing model-based design techniques can also help designers recognize and accommodate the unique contexts and context changes that are present in mobile computing. Without an abstract description of a user interface, the development of mobile user interfaces can be error-prone and time-consuming.

Quote:

"Mobile apps force designers to deeply understand user needs so as to provide just the right functions in a learnable and useable interface."

Ben Schneiderman

13 Brand, ethical preferences, moral preferences, cognitive beliefs.

INFO

MobileApp User Interface Design Considerations

Design choices affect performance and should be examined early in the UI design process. Ivo Weevers [Wee11] posted several mobile user–interface design practices that have proven to be helpful when designing mobile applications:

- **Define user interface brand signatures**. Differentiate the app from its competitors. Make the core signature elements of the brand the most responsive, since users will use them over and over.
- **Focus the portfolio of products**. Target the platforms that are most important to the success of the app and the company. Not all platforms have the same number of users.
- **Identify the core user stories**. Make use of techniques such as quality function deployment (Chapter 8) to reduce a lengthy list of requirements to implement using the constrained resources available on mobile devices.
- **Optimize UI flows and elements**. Users do not like to wait. Identify potential bottlenecks in user work flow and make sure the user is given indication of progress when delays occur. Make sure that the time to display screen elements is justified in terms of user benefits.
- **Define scaling rules.** Determine the options that will be used when information to be displayed is too large to fit on the screen. Managing functionality, aesthetics, usability, and performance is a continual balancing act.
- **Use a performance dashboard.** Used to communicate the product's current state of completion (e.g., number of use stories implemented), its performance relative to its targets, and perhaps comparisons to its competitors.
- **Champion dedicated UI engineering skills.** It is important to understand that the implementation of layout, graphics, and animation has performance implications. Techniques to interleave rendering of display items and program execution can be helpful.

18.2.3 Context-Aware Apps

Context allows the creation of new applications based on the location of the mobile device and the functionality to be delivered by the device. Context can also help tailor personal computer applications for mobile devices (e.g., downloading patient information to a device carried by a home health care worker as he arrives at the patient's house).

Using highly adaptive, contextual interfaces is a good way to deal with device limitations (e.g., screen size and memory). To facilitate the development of context-aware user interaction requires support of corresponding software architectures.

In an early discussion of context-aware applications, Rodden [Rod98] points out that mobile computing merges the real and virtual worlds by providing functionality that allows a device to be aware of its location, time, and other objects in its surroundings. The device could be in a fixed location like an alarm sensor, embedded in an autonomous device, or be carried around by a human. Because the device can be designed to be used by individuals, groups, or the public, it must detect the presence and identity of the user, as well as the attributes of the context that are relevant or permitted for that user (even if the user is another device).

? How does a design achieve context awareness?

To achieve context awareness, mobile systems must produce reliable information in the presence of uncertain and rapidly changing data from a variety

of heterogeneous sources. Extracting relevant context information by combing data from several sensors proves challenging because of problems with noise, miscalibration, wear and tear, and weather. Event-based communication is preferable to the management of continuous streams of high-abstraction-level data in context-aware applications [Kor03].

In ubiquitous computing environments, multiple users work with a wide range of different devices. The configuration of the devices should be flexible enough to change frequently because of the demands of mobile work practices. It is important for the software infrastructure to support different styles of interaction (e.g., gestures, voice, and pen) and store them in abstractions that can be shared easily.

There are times when one user may desire to work with more than one device simultaneously on the same product (e.g., use a touch-screen device to edit a document image and a personal keyboard to edit document text). It is challenging to integrate mobile devices that are not always connected to the network and have a variety of device constraints [Tan01]. Networked, multiplayer games have had to deal with these problems by storing the game state on each device and sharing change information among other game players' devices in real time.

18.2.4 Lessons Learned

Earlier in this chapter we noted a number of important differences between MobileApps and conventional software. As a consequence of these differences, software engineers should modify and extend conventional techniques in order to analyze, design, build, and test mobile applications. de Sá and Carrico [Des08] suggest a number of lessons learned.

ADVICE

Use cases can work well in the development of MobilApps, but context variables must be considered when developing them.

Usage scenarios (Chapter 15) for MobileApps must consider context variables (location, user, and device) and transitions between contextual scenarios (e.g., user moves from bedroom to kitchen or switches from stylus to a finger). de Sá and Carriço have identified a set of scenario-variable types that should be considered in developing the user scenarios—locations and settings, movement and posture, devices and usages, workloads and distractions, user preferences.

Ethnographic observation[14] is a widely used method for gathering information about representative users of a software product as it is being designed. It is often difficult to observe users as they change contexts, because the observer must follow users for long periods of time, something that could raise privacy concerns.[15] A complicating factor is that users sometimes complete tasks differently in private settings than in social settings. The same users may need to be observed performing tasks in multiple contexts while monitoring transitions, as well as recording user reactions to the changes.

14 *Ethnographic observation* is a means determining the nature of user tasks by watching users in their work environment.

15 Asking users to fill out anonymous questionnaires may have to suffice when direct observation is not possible.

Early user interface prototypes for MobileApps can be created on paper, using sketches or a combination of index cards and/or post-it notes to emulate important interaction mechanisms. The key is to allow all interaction mechanisms to be assessed, all usage contexts to be examined, and all user interaction mechanisms to be specified. In addition, the use of interaction widgets and overall screen placement and location can also be emulated. These early rough paper prototypes can assist in uncovering errors, inconsistencies, and omissions before targeted mobile devices come into play. Later prototypes may then be created to run on targeted mobile devices once the layout and placement issues have been resolved.

INFO

MobileApp Design Mistakes

Joh Koester [Koe12] posts several examples of MobileApp design practices that should be avoided:

- **Kitchen Sink**. Avoid adding too many features to the app and too many widgets on the screen. Simple is understandable. Simple is marketable.
- **Inconsistency**. To avoid this, set standards for page navigation, menu use, buttons, tabs, and other user-interface elements. Stick to a uniform look and feel.
- **Overdesigning.** Be ruthless when designing a MobileApp. Remove unnecessary elements and wasteful graphics. Do not be tempted to add elements just because you think you should.
- **Lack of Speed**. Users do not care about device constraints—they want to view things quickly. Preload what you can. Eliminate what is not needed.
- **Verbiage**. Unnecessarily long, wordy menus and screen displays are indications of a MobileApp that has not been tested with users and developers who have not spent enough time understanding the user's task.
- **Nonstandard Interaction.** One reason for targeting a platform is to take advantage of the user's experience with the ways things are done on that platform. Where standards exist use them. This needs to be balanced with the need to have an application appear and behave the same way on multiple devices when possible.
- **Help-and-FAQ-itis.** Adding online help is not the way to repair a poorly designed user interface. Make sure you have tested your app with your targeted users and repaired the identified defects.

18.3 MobileApp Design—Best Practices

There are several guidelines for developing MobileApps[16] and for developing apps for specific platforms like Apple's iOS[17] or Google's Android.[18] Schumacher [Sch09] has collected many best practice ideas and has posted several specially adapted to the design of a mobile application as Web pages.[19] Some important

16 **http://www.w3.org/TR/mwabp/**

17 **https://developer.apple.com/library/iOS/navigation/**

18 **http://developer.android.com/guide/components/index.html**

19 **http://www.usercentric.com/news/2011/06/15/best-practices-designing-mobile-touch-screen-applications**

considerations when designing mobile touch-screen applications listed by Schumacher include:

? What are the things I should think about when designing mobile touch-screen apps?

- **Identify your audience.** The application must be written with the expectations and backgrounds of its users in mind. Experienced users want to do things quickly. Less experienced users will appreciate a handholding approach when they are first using the app.
- **Design for context of use.** It is important to consider how the user will interact with the real world while using the MobileApp. Watching a movie on an airplane calls for a different user interface than checking the weather before you leave the office.
- **There is a fine line between simplicity and laziness.** Creating an intuitive user interface on a mobile device is much harder than simply removing features found in the user interface for the application running on a larger device. The user interface should provide all the information that enables a user to make her next decision.
- **Use the platform as an advantage.** Touch-screen navigation is not intuitive and must be learned by all new users. This learning task will be easier if the user interface designers adhere to standards that have been set for the platform.
- **Make scrollbars and selection highlighting more salient.** Scrollbars are often hard to locate on touch devices because they are too small. Make sure that menu or icon borders are wide enough for color changes to catch the users' attention. When color coding is used, make sure there is sufficient contrast between foreground and background colors to allow them to be distinguishable by any colorblind users.
- **Increase discoverability of advanced functionality.** Hot keys and other shortcuts are sometimes included in MobileApps to allow experienced users to complete their tasks more quickly. You can increase the discoverability of features like these by including visual design clues in the user interface.
- **Use clear and consistent labels.** Widget labels should be recognized by all app users, regardless of standards used by specific platforms. Use abbreviations cautiously and avoid them if possible.
- **Clever icons should never be developed at the expense of user understanding.** Icons sometimes only make sense to their designers. Users must be able to learn their meaning quickly. It is hard to guarantee that icons are meaningful across all languages and user groups. A good strategy to enhance recognition is to add a text label beneath a novel icon.

- **Support user expectations for personalization.** Mobile device users expect to be able to personalize everything. At the very least, developers should try to allow users to set their location (or detect it automatically) and select content options that may be available at that location. It is important to indicate to users what features can be personalized and how users can personalize them.
- **Long scrolling forms trump multiple screens on mobile devices.** Experienced mobile device users want all information on a single input screen even if this requires scrolling. Novice users often become experienced quickly and will grow tired of multiple input screens.

Developing native applications for multiple device platforms can be costly and time-consuming. Development costs can be reduced by using technologies familiar to Web developers (e.g., JavaScript, CSS, and HTML) to create MobileApps that will be accessed using a Web browser on the mobile device. Open webOS[20] is a device-independent platform intended to allow this type of development.

18.4 Mobility Environments

The sidebar contains pointers to several tools that can be used to develop MobileApps for popular platforms. Each has its own advantages and disadvantages.[21] Some use technologies that are restricted to a single manufacturer's devices (e.g., iOS and Objective C). Some platforms are licensed to several manufacturers (e.g., Android and Java or Windows 8 and C#). Some are open source and designed to work on many devices (e.g., webOS and Enyo). Each platform has its own rules for marketing and distribution and each varies in the degree to which it supports specific application technologies such as gaming.

Choosing a platform (or platforms) requires careful thought by mobile developers. Sometimes the platform(s) chosen will be dictated by the customer's business goals. In other situations platform choices will be determined by the device features they support or hardware limitations that exist. Yuan [Yua02] uses the following criteria to assess several *mobile interactive development environments* (MIDEs):

? How should I go about choosing mobility environments and platforms?

- **General productivity features**. The MIDE should contain tools to support editing, project management, debugging, architectural design, documentation, and unit testing.

20 Information on webOS can be found at **https://developer.palm.com**

21 Further discussion can be found at **http://www.cs.colorado.edu/~kena/classes/5828/s10/presentations/software_engineering_mobile.pdf**

INFO

MobileApp Development Tools

Many of the tools to develop MobileApps for popular devices are available as free downloads from the World Wide Web:[22]

https://developer.apple.com/resources—Apple's iOS Dev Center contains tools that can be used to develop apps for the iPod, iPad, and iPhone.

http://developer.android.com/index.html—This site provides a plug-in to allow Android development using the Eclipse programming environment.

https://developer.blackberry.com/java/download/eclipse/—This site provides a plug-in to allow BlackBerry development using the Eclipse programming environment.

http://eclipse.org/—Download site for the Eclipse Programming Environment.

http://create.msdn.com/en-US/—Microsoft tools to develop Windows Phone apps and Xbox games.

http://www.developer.nokia.com/Develop/Java/Tools/Series_40_platform_SDKs/—Download site for several Java-based Nokia development tools.

https://developer.palm.com/content/resources/—Download site for HP webOS development environment.

http://enyojs.com/—Download site for the Enyo cross platform development environment.

http://www.scirra.com/construct2—Download site for the Construct2 cross platform game development environment.

- **Third-party SDK integration**. Each network or cloud service is likely to require the use of a specific API or SDK. It is easier to continue working in the one IDE, rather than several.
- **Post-compilation tools**. An effective MIDE contains tools that allow the source code for a completed app to be optimized for a specific mobile device or service.
- **Over-the-air deployment support**. A good MIDE should allow the testing of the deployed app within the development environment. This can be tricky when the MobileApp needs to access Web services or other applications.
- **End-to-end mobile application development**. Mobile devices are often not powerful enough to process or store large amounts of information locally. It is important to allow developers to create, test, and deploy entire mobile projects using a desktop MIDE.
- **Documentation and tutorials**. Even free development tools need to be easy to learn and easy to use. Having adequate support materials is essential.
- **Graphical user interface builders**. If the MIDE supports visual construction of user screens, prototypes can be constructed and tested quickly.

As we have already noted, it is difficult to design and implement one MobileApp that runs seamlessly on multiple platforms. This is especially true for

22 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

applications in mobile game development (a multibillion-dollar industry). Most popular games are developed in parallel for several mobile devices. This type of development fragmentation drives up costs and underscores the need for better standardization of development tools and APIs. Galavas [Gal11] notes that portability, functionality, development speed, and performance are key selection criteria when considering which mobile development platforms to use.

Mobile computing middleware can be used to facilitate the communication and coordination of distributed system components. This can allow mobile application developers to rely on abstractions that hide the details of some of the complexities of mobile environments. For middleware to be useful in MobileApp development, both the mobile client and the mobile service provider must allow for asynchronous intermittent connections. The middleware components running on the mobile client must not consume significant computational resources on the mobile device. The middleware must also help the mobile application achieve the level of context awareness required by its users [Mas02].

INFO

MobileApp Middleware

The following middleware products are representative of those developed specifically for mobile applications:

http://www.infrae.com/products/mobi—Mobi mobile middleware is a set of libraries and WSGI components that interact between a Web server and applications that make mobile data available.

http://smartsoftmobile.com/—SmartSoft Mobile Solutions provides cloud-based and enterprise (e.g., SAP) solutions for mobile device platforms.

http://www.sybase.com/—Sybase provides a Mobile Enterprise Application Platform (MEAP) that provide tools and client-server middleware for mobile and enterprise application development. See also: **http://scn.sap.com/community/mobile**

http://code.google.com/p/skeenzone/—SkeenZone is lightweight and extensible Java middleware that enables development of distributed mobile applications.

http://modolabs.com/platform—Kurogo is an open-source platform designed to power content-rich, multifaceted mobile websites and iPhone and Android apps.

18.5 The Cloud

Services computing[23] and cloud computing[24] enable the rapid development of large-scale distributed applications [Yau11]. These computing paradigms have made it easier and more economical to create applications on many different devices (personal computers, smartphones, and tablets). The two paradigms

23 *Services computing* focuses on architectural design and enables application development through service discovery and composition.

24 *Cloud computing* focuses on the effective delivery of services to users through flexible and scalable resource virtualization and load balancing,

Quote:

"Service-oriented software engineering incorporates the best features of both the services and cloud computing paradigms."

Stephan Yau

allow resource outsourcing and transfer information of information technology management to service providers while at the same time mitigating the impact of resource limitations on some mobile devices. A service-oriented architecture provides the architectural style (e.g., REST),[25] standard protocols (e.g., XML[26], SOAP[27]), and interfaces (e.g., WSDL)[28] needed for MobileApp development. Cloud computing enables convenient, on-demand network access to a shared pool of configurable computing resources (servers, storage, applications, and services).

Services computing allows MobileApp developers to avoid the need to integrate service source code into the client running on a mobile device. Instead, the service runs out of the provider's server and is loosely coupled with the applications that make use of it through messaging protocols. A service typically provides an API (application programming interface) to allow it to be treated like an abstract black box.

Quote:

"Because users' businesses rely heavily on third-party service providers, there are serious concerns about how threats to service reliability could affect a service and consequently a cloud user's business."

Stephan Yau

Cloud computing lets the client (either a user or program) request computing capabilities as needed, across network boundaries anywhere or any time. The cloud architecture has three layers, each of which can be called as a service. The *software as service* layer consists of software components and applications hosted by third-party service providers. The *platform as service* layer provides a collaborative development platform to assist with design, implementation, and testing by geographically distributed team members. *Infrastructure as a service* provides virtual computing resources (storage, processing power, network connectivity) on the cloud.

Mobile devices can access cloud services from any location at any time. The risks of identity theft and service hijacking require providers of mobile services and cloud computing to employ rigorous security engineering techniques to protect their users. Security and privacy concerns associated with cloud computing are discussed in Chapter 27. Using a vendor-neutral cloud service may make it easier to create cross-platform applications [Rat12].

Taivalsaari [Tai12] points out that making use of cloud storage can allow any mobile device or software features to be updated easily on millions of devices worldwide. In fact, it is possible to virtualize the entire mobile user experience so that all applications are downloaded from the cloud.

---

25 *Representation State Transfer* describes a networked web architectural style where the resource representation (e.g. a Web page) places the client in a new state. The client changes or transfers state with each resource representation.

26 *Extensible Markup Language* XML is designed to store and transport data, while HTML is designed to display data.

27 *Simple Object Access Protocol* is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.

28 *Web Services Description Language* is an XML-based language for describing Web services and how to access them.

18.6 The Applicability of Conventional Software Engineering

There are no guarantees that a desktop program or a WebApp can be easily adapted for implementation as a MobileApp. However, many of the agile software engineering practices (Chapter 5) used to create desktop computer applications can be used to create standalone MobileApps or mobile client software, and many of the practices used to create quality WebApps apply to the creation of Web services used by MobileApps.

During formulation a set of goals and user stories are assembled following practices used in many agile process models. These will define the required user experience and determine stakeholder needs that must be satisfied and the contextual variables that must be taken into account by the MobileApp. The role of context and location awareness is not likely to be considered when establishing goals for a desktop or Web application.

During the planning activity, the difficulties of developing for more than one device or platform must be factored into the project budget and timeline so that the resources needed to satisfy all stakeholder concerns are appropriately allocated. The difficulties of conducting meaningful usability testing and adequate field testing add to the development costs of MobileApps. Risk analysis should consider the impact of losses if security incidents or privacy violations occur. Planning for a security review of the requirements might also be desirable (Chapter 27).

In MobileApp development, time to market is critical. In addition, new technology elements and changing user requirements are often introduced as development proceeds. As we noted earlier, an iterative agile process model (Chapters 4 and 5) provides opportunities for developers to make adjustments to requirements based on assessments of the evolving product prototype.

During product engineering, content analysis and design are similar to actions applied when a WebApp (Chapter 17) is built. The content to be included in the MobileApp needs to be selected and chunked subject to the limitations of the targeted devices and platforms.

Design of the MobileApp can be expedited by using rapidly expanding set design patterns (Chapter 13) and component-based design (Chapter 14) oriented toward MobileApps [Mes08]. A composition strategy using component-based and object-oriented design is applied when existing services are incorporated into a MobileApp. Reuse without compromising the quality of services is a core objective in MobileApp development [Zha05].

User interface design draws heavily on the lessons learned in the graphic and aesthetic design for Web pages (Chapter 17) to support the branding goals for the MobileApp. User-centered design, with its emphasis on usability and accessibility is important to creating quality user interfaces for MobileApps (Chapter 15).

The most important architectural design decision is often whether to build a thin or fat client. The model-view-controller (MVC) architecture (Chapter 17) is commonly used in MobileApps.. Because the MobileApp architecture may have a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design. The architectural design must take device resources in to account (storage, processor speed, and network connectivity). The design should include provisions for discoverable services and movable devices.

Usability testing and deployment testing take place during each prototype development cycle. Code reviews that focus on security issues should be included as part of the implementation activities. These code reviews should be based on the appropriate security objectives and threats identified in the system design activities. Security testing is a routine part of system testing (Chapter 22).

18.7 Summary

The quality of a MobileApp—defined in terms of functionality, reliability, usability, efficiency, security, maintainability, scalability, and portability—is introduced during design. A good MobileApp should be based on the following design goals: simplicity, ubiquity, personalization, flexibility, and localization.

Interface design describes the structure and organization of the user interface and includes a representation of screen layout, a definition of the modes of interaction, and a description of navigation mechanisms. In addition, the interface for a good MobileApp will promote the brand signature and focus on its targeted device platform(s). A set of core user stories is used to trim unnecessary features from the app to manage its resources requirements. Context-aware devices make use of discoverable services to help personalize the user experience.

Content design is critically important and takes the screen and other limitations of mobile devices into account. Aesthetic design, also called graphic design, describes the "look and feel" of the MobileApp and includes color schemes; graphic layout; the use of graphics; and related aesthetic decisions. Aesthetic design must also take device limitations into account.

Architecture design identifies the overall hypermedia structure for the MobileApp and encompasses both content architecture and MobileApp architecture. It is critical to determine how much of the MobileApp functionality will reside on the mobile device and how much will be provided by Web or cloud services.

Navigation design represents the navigational flow between content objects and for all MobileApp functions. Navigation syntax is defined by the widgets available on the targeted mobile device(s) and the semantics are often determined by the mobile platform. Content chunking must take intermittent service interruptions into account and user demands for fast performance.

Component design develops the detailed processing logic required to implement the components that are used to build a complete MobileApp function. Design techniques described in Chapter 14 may be applicable for the engineering of MobileApp components.

Problems and Points to Ponder

18.1 Explain why deciding to develop MobileApp for several devices can be a costly design decision. Is there a way to mitigate the risks of supporting the wrong platform?

18.2 In this chapter we listed many quality attributes for MobileApps. Select the three that you believe are most important, and make an argument that explains why each should be emphasized in MobileApp design work.

18.3 Add at least five additional questions to the MobileApp Design—Quality Checklist presented in Section 18.2.

18.4 You are a MobileApp designer for *Project Planning Corporation*, a company that builds productivity software. You want to implement the equivalent of a digital three-ring binder that allows tablet users to organize and categorize electronic documents of several types under user-defined tabs. For example, a kitchen remodeling project might require a pdf catalog, a jpg or dfx layout drawing, an MS Word proposal, and an Excel spreadsheet stored under a Cabinetry tab. Once defined, the binder and its tab content can be stored either on the tablet or on some cloud storage. The application needs to provide five key functions: binder and tab definition, digital document acquisition from a Web location or the device, binder management functions, page display functions, and a notes function to allow a Post-it note to be added to any page. Develop an interface design for the three-ring application and implement it as a paper prototype.

18.5 What is the most aesthetically pleasing MobileApp you have ever used and why?

18.6 Create user stories for the three-ring application described in Problem 18.4.

18.7 What might be considered to make the three-ring application a context-aware MobileApp?

18.8 Reconsidering the *ProjectPlanning* three-ring application described in Problem 18.4, select a development platform for the first working prototype. Discuss why you made the choice.

18.9 Use UML to develop design representations for the interface objects that would be encountered as the three-ring application described in Problem 18.4 is designed.

18.10 Do a bit of additional research on the MVC architecture and decide whether it would be an appropriate MobileApp architecture for the three-ring discussed in Problem 18.4.

18.11 Describe three context-aware features that would be desirable to add to a *SafeHome* MobileApp.

18.12 Do some Internet research to identify a middleware product designed to support MobileApps. Describe the middleware features and the platform(s) it supports.

Further Readings and Information Sources

Kumar and Xie (*Handbook of Mobile Systems Applications and Services,* Auerbach Publications, 2012) have edited a book that covers mobile services and the role of service-oriented architectures in mobile computing. Books on pervasive computing by Adelstein

(*Fundamentals of Mobile and Pervasive Computing*, McGraw-Hill, 2004), Hansmann (*Pervasive Computing: The Mobile World*, 2nd ed., Springer, 2003 reprinted 2013) define the principles of context in mobile computing. Books by Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design*, Morgan Kaufman, 2010) and Polstad (*Ubiquitous Computing: Smart Devices, Environments and Interactions*, Wiley, 2009) describes the context-aware computing in terms of the interactions among smart devices, smart environments, and smart interactions. Neil (*Mobile Design Pattern Gallery*, O'Reilly, 2012) documents mobile design patterns illustrated by examples from six mobile platforms.

Interface design books are plentiful. A good general reference is the book edited by Schumacher (*Handbook of Global User Research*, Morgan-Kaufmann, 2009). Hoober (*Designing Mobile Interfaces*, O'Reilly, 2011) describes device-independent best practices for composing pages, displaying information, and using sensors. Nielsen (*Mobile Usability*, New Riders, 2012) offers advice on how to design usable interfaces that take mobile-device screen size into account. Colborne (*Simple and Usable Web, Mobile, and Interaction Design*, New Riders, 2010) describes the process of simplifying user interaction. Ginsburg (*Designing the iPhone User Experience: A User-Centered Approach to Sketching and Prototyping iPhone Apps*, Addison-Wesley, 2010) discusses the importance of differentiating your app from the competition by taking a user-centric approach to designing a high-quality user experience.

The *Microsoft Application Architecture Guide* (Microsoft Press, 2009) contains useful information about mobile application design and architecture. Architecture design for tablet devices is discussed in a book by Lee (*Mobile Applications: Architecture, Design, and Development*, Prentice Hall, 2004). Esposito (*Architecting Mobile Solutions for the Enterprise*, Microsoft Press, 2012) describes the process of developing a cross-platform application by building an effective mobile website. Mobile middleware is discussed in a book edited by Garbinato (*Middleware for Network Eccentric and Mobile Applications*, Springer, 2009).

There are many books on MobileApp programming that focus on a specific platform. Books by Firtman (*Programming the Mobile Web,* O'Reilly, 2010). Mednieke (*Programming Android,* O'Reilly, 2011) or Lee (*Test-Driven iOS Development,* Addison-Wesley, 2012) are representative.

A wide variety of information sources on design for MobileApps is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

PART

Three

QUALITY MANAGEMENT

In this part of *Software Engineering: A Practitioner's Approach* you'll learn about the principles, concepts, and techniques that are applied to manage and control software quality. These questions are addressed in the chapters that follow:

- What are the generic characteristics of high-quality software?
- How do we review quality and how are effective reviews conducted?
- What is software quality assurance?
- What strategies are applicable for software testing?
- What methods are used to design effective test cases?
- Are there realistic methods that will ensure that software is correct?
- How can we manage and control changes that always occur as software is built?
- What measures and metrics can be used to assess the quality of requirements and design models, source code, and test cases?

Once these questions are answered you'll be better prepared to ensure that high-quality software has been produced.

CHAPTER

19 Quality Concepts

Key Concepts

cost of quality.... 422
good enough..... 421
liability..... 425
management actions..... 426
quality..... 413
quality dilemma..... 420
quality dimensions..... 415
quality factors ... 416
quantitative view 420
risks..... 424
security 425

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised. Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, *CIO Magazine* trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do" [Lev01]. *InformationWeek* [Ric01] echoed the same concern:

> Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn't include the cost of losing angry customers. Because IT shops write applications that rely on packaged infrastructure software, bad code can wreak havoc on custom apps as well . . .
>
> Just how bad is bad software? Definitions vary, but experts say it takes only three or four defects per 1,000 lines of code to make a program perform poorly. Factor in that most programmers inject about one error for every 10 lines of code

Quick Look

What is it? The answer isn't as easy as you might think. You know quality when you see it, and yet, it can be an elusive thing to define. But for computer software, quality is something that we must define, and that's what we'll do in this chapter.

Who does it? Everyone—software engineers, managers, all stakeholders—involved in the software process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time to market.

What are the steps? To achieve high-quality software, four activities must occur: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure.

What is the work product? Software that meets its customer's needs, performs accurately and reliably, and provides value to all who use it.

How do I ensure that I've done it right? Track quality by examining the results of all quality control activities, and measure quality by examining errors before delivery and defects released to the field.

> they write, multiply that by the millions of lines of code in many commercial products, then figure it costs software vendors at least half their development budgets to fix errors while testing. Get the picture?

In 2005, *ComputerWorld* [Hil05] lamented that "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction. A year later, *InfoWorld* [Fos06] wrote about the "the sorry state of software quality" reporting that the quality problem had not gotten any better. As the emphasis on software quality grew, a survey of 100,000 white-collar professionals [Rog12] indicated that software quality engineers were "the happiest workers in America"!

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both*—and that's the problem. In this chapter, we consider software quality as a concept and examine why it's worthy of serious consideration whenever software engineering practices are applied.

19.1 What Is Quality?

In his mystical book *Zen and the Art of Motorcycle Maintenance,* Robert Persig [Per74] commented on the thing we call *quality*:

> Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others; that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Indeed—what is it?

? What are the different ways in which quality can be viewed?

At a somewhat more pragmatic level, David Garvin [Gar84] of the Harvard Business School suggests that "quality is a complex and multifaceted concept" that can be described from five different points of view. The *transcendental view* argues (like Persig) that quality is something you immediately recognize, but cannot explicitly define. The *user view* sees quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality. The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms

Quote:

"People forget how fast you did a job—but they always remember how well you did it."

Howard Newton

to the spec, it exhibits quality. The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product. Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases if the product is manufactured according to specifications.

In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass [Gla98] argues that a more "intuitive" relationship is in order:

user satisfaction = compliant product + good quality + delivery within budget and schedule

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco [DeM98] reinforces this view when he states: "A product's quality is a function of how much it changes the world for the better." This view of quality contends that if a software product provides substantial benefit to its end users, they may be willing to tolerate occasional reliability or performance problems.

19.2 Software Quality

? How can I best define software quality?

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *software* quality? In the most general sense, software quality can be defined as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*[1]

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. An *effective software process* establishes the infrastructure that supports any effort at building a high-quality software product. The management

1 This definition has been adapted from [Bes04] and replaces a more manufacturing-oriented view presented in earlier editions of this book.

Quote:

"Be a yardstick of quality. Some people aren't used to an environment where excellence is expected."

Steve Jobs

aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high-quality software. Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

2. A *useful product* delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software.
3. By *adding value for both the producer and user* of a software product, high-quality software provides benefits for the software organization and the end-user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process. The end result is (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial for the business.

19.2.1 Garvin's Quality Dimensions

ADVICE

You can use a radar diagram to provide a visual representation of each of Garvin's quality dimensions as they are applied to an app.

David Garvin [Gar87] suggests that quality should be considered by taking a multidimensional viewpoint that begins with an assessment of conformance and terminates with a transcendental (aesthetic) view. Although Garvin's eight dimensions of quality were not developed specifically for software, they can be applied when software quality is considered:

Performance Quality. Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end user?

Feature quality. Does the software provide features that surprise and delight first-time end users?

Reliability. Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?

Conformance. Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto

design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Durability. Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?

Serviceability. Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period? Can support staff acquire all information they need to make changes or correct defects? Douglas Adams [Ada93] makes a wry comment that seems appropriate here: "The difference between something that can go wrong and something that can't possibly go wrong is that when something that can't possibly go wrong goes wrong it usually turns out to be impossible to get at or repair."

Aesthetics. There's no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.

Perception. In some situations, you have a set of prejudices that will influence your perception of quality. For example, if you are introduced to a software product that was built by a vendor who has produced poor quality in the past, your guard will be raised and your perception of the current software product quality might be influenced negatively. Similarly, if a vendor has an excellent reputation, you may perceive quality, even when it does not really exist.

Garvin's quality dimensions provide you with a "soft" look at software quality. Many (but not all) of these dimensions can only be considered subjectively. For this reason, you also need a set of "hard" quality factors that can be categorized in two broad groups: (1) factors that can be directly measured (e.g., defects uncovered during testing) and (2) factors that can be measured only indirectly (e.g., usability or maintainability). In each case measurement must occur. You should compare the software to some datum and arrive at an indication of quality.

19.2.2 McCall's Quality Factors

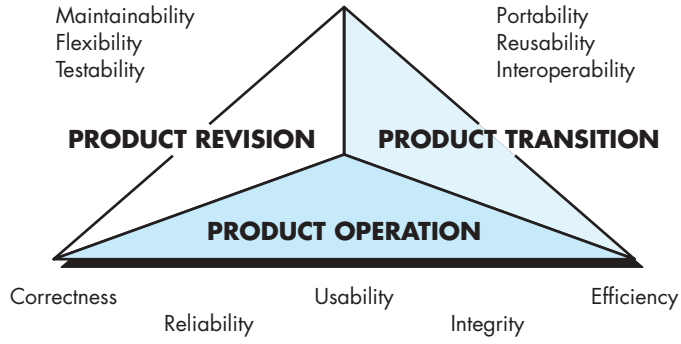
McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 19.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

Referring to the factors noted in Figure 19.1, McCall and his colleagues provide the following descriptions:

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

FIGURE 19.1

McCall's software quality factors



Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 21).]

Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. Extent to which access to software or data by unauthorized persons can be controlled.

Usability. Effort required to learn, operate, prepare input for, and interpret output of a program.

Maintainability. Effort required to locate and fix an error in a program. [This is a very limited definition.]

Flexibility. Effort required to modify an operational program.

Testability. Effort required to test a program to ensure that it performs its intended function.

Portability. Effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

Interoperability. Effort required to couple one system to another.

Quote:

"The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten."

Karl Weigers (unattributed quote)

It is difficult, and in some cases impossible, to develop direct measures[2] of these quality factors. In fact, many of the metrics defined by McCall and colleagues can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality.

2 A *direct measure* implies that there is a single countable value that provides a direct indication of the attribute being examined. For example, the "size" of a program can be measured directly by counting the number of lines of code.

19.2.3 ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

ADVICE

Although it's tempting to develop quantitative measures for the quality factors noted here, you can also create a simple checklist of attributes that provide a solid indication that the factor is present.

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

Like other software quality factors discussed in the preceding subsections, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.

19.2.4 Targeted Quality Factors

Quote:

"Any activity becomes creative when the doer cares about doing it right, or better."

John Updike

The quality dimensions and factors presented in Sections 19.2.1 and 19.2.2 focus on the software as a whole and can be used as a generic indication of the quality of an application. A software team can develop a set of quality characteristics and associated questions that would probe the degree to which each factor has been satisfied.[3] For example, McCall identifies *usability* as an important quality factor. If you were asked to review a user interface and assess its usability, how would you proceed? You might start with the subattributes suggested by McCall—understandability, learnability, and operability—but what do these mean in a pragmatic sense?

3 These characteristics and questions would be addressed as part of a software review (Chapter 20).

To conduct your assessment, you'll need to address specific, measurable (or at least, recognizable) attributes of the interface. For example [Bro03]:

Intuitiveness. The degree to which the interface follows expected usage patterns so that even a novice can use it without significant training.

- Is the interface layout conducive to easy understanding?
- Are interface operations easy to locate and initiate?
- Does the interface use a recognizable metaphor?
- Is input specified to economize key strokes or mouse clicks?
- Does the interface follow the three golden rules? (Chapter 15)
- Do aesthetics aid in understanding and usage?

Efficiency. The degree to which operations and information can be located or initiated.

- Does the interface layout and style allow a user to locate operations and information efficiently?
- Can a sequence of operations (or data input) be performed with an economy of motion?
- Are output data or content presented so that it is understood immediately?
- Have hierarchical operations been organized in a way that minimizes the depth to which a user must navigate to get something done?

Robustness. The degree to which the software handles bad input data or inappropriate user interaction.

- Will the software recognize the error if data values are at or just outside prescribed input boundaries? More importantly, will the software continue to operate without failure or degradation?
- Will the interface recognize common cognitive or manipulative mistakes and explicitly guide the user back on the right track?
- Does the interface provide useful diagnosis and guidance when an error condition (associated with software functionality) is uncovered?

Richness. The degree to which the interface provides a rich feature set.

- Can the interface be customized to the specific needs of a user?
- Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command?

As the interface design is developed, the software team would review the design prototype and ask the questions noted. If the answer to most of these questions is yes, it is likely that the user interface exhibits high quality.

A collection of questions similar to these would be developed for each quality factor to be assessed.

19.2.5 The Transition to a Quantitative View

In the preceding subsections, we have presented a set of qualitative factors for the "measurement" of software quality. The software engineering community strives to develop precise measures for software quality and is sometimes frustrated by the subjective nature of the activity. Cavano and McCall [Cav78] discuss this situation:

> The determination of quality is a key factor in everyday events—wine tasting contests, sporting events [e.g., gymnastics], talent contests, etc. In these situations, quality is judged in the most fundamental and direct manner: side by side comparison of objects under identical conditions and with predetermined concepts. The wine may be judged according to clarity, color, bouquet, taste, etc. However, this type of judgment is very subjective; to have any value at all, it must be made by an expert.
>
> Subjectivity and specialization also apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as a way to derive quantitative measurements of software quality for objective analysis . . . Since there is no such thing as absolute knowledge, one should not expect to measure software quality exactly, for every measurement is partially imperfect. Jacob Bronkowski described this paradox of knowledge in this way: "Year by year we devise more precise instruments with which to observe nature with more fineness. And when we look at the observations we are discomfited to see that they are still fuzzy, and we feel that they are as uncertain as ever."

ADVICE

When you're faced with the quality dilemma (and everyone is faced with it at one time or another), try to achieve balance—enough effort to produce acceptable quality without burying the project.

In Chapter 30, we'll present a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure *quality* but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

19.3 The Software Quality Dilemma

In an interview [Ven03] published on the Web, Bertrand Meyer discusses what I call the *quality dilemma:*

> If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway. Either you missed the market window, or you simply exhausted all your resources. So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

ADVICE

Although it's tempting to develop quantitative measures for the quality factors noted here, you can also create a simple checklist of attributes that provide a solid indication that the factor is present.

It's fine to state that software engineers should strive to produce high-quality systems. It's even better to apply good practices in your attempt to do so. But the situation discussed by Meyer is real life and represents a dilemma for even the best software engineering organizations.

19.3.1 "Good Enough" Software

Stated bluntly, if we are to accept the argument made by Meyer, is it acceptable to produce "good enough" software? The answer to this question must be yes, because major software companies do it every day. They create software with known bugs and deliver it to a broad population of end users. They recognize that some of the functions and features delivered in Version 1.0 may not be of the highest quality and plan for improvements in Version 2.0. They do this knowing that some customers will complain, but they recognize that time to market may trump better quality as long as the delivered product is "good enough."

Exactly what is "good enough"? Good enough software delivers high-quality functions and features that end users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality.

This idea may resonate with many readers. If you're one of them, we can only ask you to consider some of the arguments against "good enough."

It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in. As we noted earlier, it can argue that it will improve quality in subsequent versions. By delivering a good enough version 1.0, it has cornered the market.

If you work for a small company be wary of this philosophy. When you deliver a good enough (buggy) product, you risk permanent damage to your company's reputation. You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.

If you work in certain application domains (e.g., real-time embedded software) or build application software that is integrated with hardware (e.g., automotive software, telecommunications software), delivering software with known bugs can be negligent and open your company to expensive litigation. In some cases, it can even be criminal. No one wants good enough aircraft avionics software!

So, proceed with caution if you believe that "good enough" is a shortcut that can solve your software quality problems. It can work, but only for a few and only in a limited set of application domains.[4]

4 A worthwhile discussion of the pros and cons of "good enough" software can be found in [Bre02].

19.3.2 The Cost of Quality

The argument goes something like this—*we know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want.* On its face, this argument seems reasonable (see Meyer's comments earlier in this section). There is no question that quality has a cost, but lack of quality also has a cost—not only to end users who must live with buggy software, but also to the software organization that has built and must maintain it. The real question is this: *which cost should we be worried about*? To answer this question, you must understand both the cost of achieving quality and the cost of low-quality software.

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization should collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

ADVICE

Don't be afraid to incur significant prevention costs. Rest assured that your investment will provide an excellent return.

Prevention costs include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities.

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include: (1) the cost of conducting technical reviews (Chapter 20) for software engineering work products, (2) the cost of data collection and metrics evaluation (Chapter 30), and (3) the cost of testing and debugging (Chapters 22 through 26).

Failure costs are those that would disappear if no errors appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when you detect an error in a product prior to shipment. Internal failure costs include: (1) the cost required to perform rework (repair) to correct an error, (2) the cost that occurs when rework inadvertently generates side effects that must be mitigated, and (3) the costs associated with the collection of quality metrics that allow an organization to assess the modes of failure. *External failure costs* are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real. Bad things happen when low-quality software is produced.

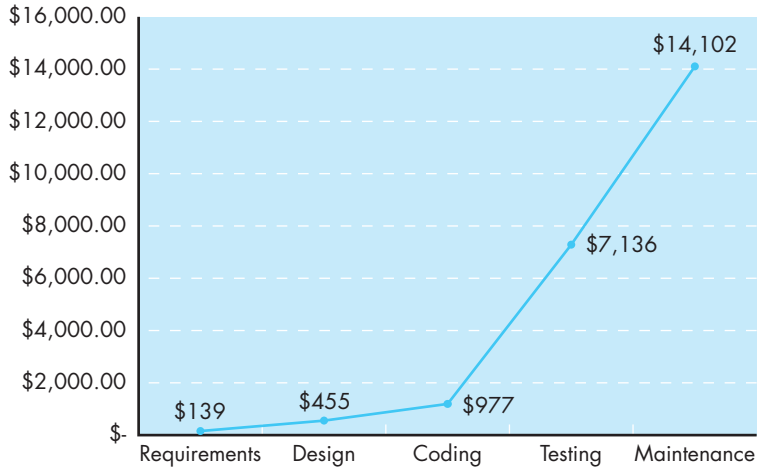
Quote:

"It takes less time to do a thing right than to explain why you did it wrong."

H. W. Longfellow

FIGURE 19.2

Relative cost of correcting errors and defects
Source: Adapted from [Boe01b].



In an indictment of software developers who refuse to consider external failure costs, Cem Kaner [Kan95] states:

> Many of the external failure costs, such as goodwill, are difficult to quantify, and many companies therefore ignore them when calculating their cost-benefit tradeoffs. Other external failure costs can be reduced (e.g. by providing cheaper, lower-quality, post-sale support, or by charging customers for support) without increasing customer satisfaction. By ignoring the costs to our customers of bad products, quality engineers encourage quality-related decision-making that victimizes our customers, rather than delighting them.

As expected, the relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 19.2, based on data collected by Boehm and Basili [Boe01b] and illustrated by Cigital Inc. [Cig07], illustrates this phenomenon.

The industry average cost to correct a defect during code generation is approximately $977 per error. The industry average cost to correct the same error if it is discovered during system testing is $7,136 per error. Cigital Inc. [Cig07] considers a large application that has 200 errors introduced during coding.

> According to industry average data, the cost of finding and correcting defects during the coding phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 × $977) is approximately $195,400.
>
> Industry average data shows that the cost of finding and correcting defects during the system testing phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 × $7,136) would have been approximately $356,800. This would also have resulted

in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 × $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been $2,472,100 ($2,115,300 + $356,800).

Even if your software organization has costs that are half of industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

SafeHome

Quality Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and other members of the product software engineering team.

The conversation:

Doug: I was looking at an industry report on the costs of repairing software defects. They are pretty sobering.

Jamie: We are already working on developing test cases for each functional requirement.

Doug: That's good, but I was noticing that it costs eight times as much to repair a defect that is discovered in testing than it does if the defect is caught and repaired during coding.

Vinod: We are using pairs programming so we should be able to catch most of the defects during coding.

Doug: I think you are missing the point. Quality is more than simply removing coding errors. We need to look at the project quality goals and ensure that the evolving software products are meeting them.

Jamie: Do you mean things like usability, security, and reliability?

Doug: Yes, I do. We need to build checks into the software process to monitor our progress toward meeting our quality goals.

Vinod: Can't we finish the first prototype and then check it for quality?

Doug: I am afraid not. We must establish a culture of quality early in the project.

Vinod: What do you want us to do, Doug?

Doug: I think we will need to find a technique that will allow us to monitor the quality of the *SafeHome* products. Let's think about this and revisit this again tomorrow.

19.3.3 Risks

Quality is built in

In Chapter 1 of this book, we wrote "people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right." The implication is that low-quality software increases risks for both the developer and the end user. In the preceding subsection, we discussed one of these risks (cost). But the downside of poorly designed and implemented applications does not always stop with dollars and time. An extreme example [Gag04] might serve to illustrate.

Throughout the month of November 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, 5 of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy?

A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.

The three Panamanian medical physicists, who tweaked the software to provide additional capability, were charged with second-degree murder. The U.S. company was faced with serious litigation in two countries. Gage and McCormick comment:

> This is not a cautionary tale for medical technicians, even though they can find themselves fighting to stay out of jail if they misunderstand or misuse technology. This also is not a tale of how human beings can be injured or worse by poorly designed or poorly explained software, although there are plenty of examples to make the point. This is a warning for any creator of computer programs: that software quality matters, that applications must be foolproof, and that—whether embedded in the engine of a car, a robotic arm in a factory or a healing device in a hospital—poorly deployed code can kill.

Poor quality leads to risks, some of them very serious.

19.3.4 Negligence and Liability

The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity. The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., health care administration or homeland security).

Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. Litigation ensues.

In most cases, the customer claims that the developer has been negligent (in the manner in which it has applied software practices) and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.

19.3.5 Quality and Security

As the criticality of Web-based and mobile systems grows, application security has become increasingly important. Stated simply, software that does not exhibit high quality is easier to hack, and as a consequence, low-quality software can indirectly increase the security risk with all of its attendant costs and problems.

In an interview in *ComputerWorld,* author and security expert Gary McGraw comments [Wil05]:

> Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process].

> Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws.

To build a secure system, you must focus on quality, and that focus must begin during design. The concepts and methods discussed in Part 2 of this book lead to a software architecture that reduces "flaws." A more detailed discussion of security engineering is presented in Chapter 27.

19.3.6 The Impact of Management Actions

Software quality is often influenced as much by management decisions as it is by technology decisions. Even the best software engineering practices can be subverted by poor business decisions and questionable project management actions.

In Part 4 of this book we discuss project management within the context of the software process. As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

Estimation decisions. A software team is rarely given the luxury of providing an estimate for a project *before* delivery dates are established and an overall budget is specified. Instead, the team conducts a "sanity check" to ensure that delivery dates and milestones are rational. In many cases there is enormous time-to-market pressure that forces a team to accept unrealistic delivery dates. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.

Scheduling decisions. When a software project schedule is established (Chapter 34), tasks are sequenced based on dependencies. For example, because component **A** depends on processing that occurs within components **B, C,** and **D,** component **A** cannot be scheduled for testing until components **B, C,** and **D** are fully tested. A project schedule would reflect this. But if time is very short, and **A** must be available for further critical testing, you might decide to test **A** without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, **A** may have defects that are hidden, only to be discovered much later. Quality suffers.

Risk-oriented decisions. Risk management (Chapter 35) is one of the key attributes of a successful software project. You really do need to know what might go wrong and establish a contingency plan if it does. Too many software teams prefer blind optimism, establishing a development schedule under the assumption

that nothing will go wrong. Worse, they don't have a way of handling things that do go wrong. As a consequence, when a risk becomes a reality, chaos reigns, and as the degree of craziness rises, the level of quality invariably falls.

The software quality dilemma can best be summarized by stating Meskimen's law —*There's never time to do it right, but always time to do it over again.* Our advice: taking the time to do it right is almost never the wrong decision.

19.4 Achieving Software Quality

Software quality doesn't just appear. It is the result of good project management and solid software engineering practice. Management and practice are applied within the context of four broad activities that help a software team achieve high software quality: software engineering methods, project management techniques, quality control actions, and software quality assurance.

19.4.1 Software Engineering Methods

If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions and factors discussed in Section 19.2.

In Part 2 of this book, we presented a wide array of concepts and methods that can lead to a reasonably complete understanding of the problem and a comprehensive design that establishes a solid foundation for the construction activity. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially.

19.4.2 Project Management Techniques

? What do I need to do to affect quality in a positive way?

The impact of poor management decisions on software quality has been discussed in Section 19.3.6. The implications are clear: if (1) a project manager uses estimation to verify that delivery dates are achievable, (2) schedule dependencies are understood and the team resists the temptation to use shortcuts, (3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.

In addition, the project plan should include explicit techniques for quality and change management. Techniques that lead to good project management practices are discussed in Part 4 of this book.

19.4.3 Quality Control

? What is software quality control?

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent. Code may be inspected in order to uncover and correct errors before testing commences. A series of testing steps

is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals. Quality control activities are discussed in detail throughout the remainder of Part 3 of this book.

19.4.4 Quality Assurance

WebRef
Useful links to SQA resources can be found at **www.niwotridge.com/Resources/PM-SWEResources/SoftwareQualityAssurance.htm.**

Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working. Of course, if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues. Software quality assurance is discussed in detail in Chapter 21.

19.5 Summary

Concern for the quality of the software-based systems has grown as software becomes integrated into every aspect of our daily lives. But it is difficult to develop a comprehensive description of software quality. In this chapter quality has been defined as an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

A wide variety of software quality dimensions and factors has been proposed over the years. All try to define a set of characteristics that, if achieved, will lead to high software quality. McCall's and the ISO 9126 quality factors establish characteristics such as reliability, usability, maintainability, functionality, and portability as indicators that quality exists.

Every software organization is faced with the software quality dilemma. In essence, everyone wants to build high-quality systems, but the time and effort required to produce "perfect" software are simply unavailable in a market-driven world. The question becomes, should we build software that is "good enough"? Although many companies do just that, there is a significant downside that must be considered.

Regardless of the approach that is chosen, quality does have a cost that can be discussed in terms of prevention, appraisal, and failure. Prevention costs include all software engineering actions that are designed to prevent defects in the first place. Appraisal costs are associated with those actions that assess software

work products to determine their quality. Failure costs encompass the internal price of failure and the external effects that poor quality precipitates.

Software quality is achieved through the application of software engineering methods, solid management practices, and comprehensive quality control—all supported by a software quality assurance infrastructure. In the chapters that follow, quality control and assurance are discussed in some detail.

Problems and Points to Ponder

19.1 Describe how you would assess the quality of a university before applying to it. What factors would be important? Which would be critical?

19.2 Garvin [Gar84] describes five different views of quality. Provide an example of each using one or more well-known electronic products with which you are familiar.

19.3 Using the definition of software quality proposed in Section 19.2, do you think it's possible to create a useful product that provides measurable value without using an effective process? Explain your answer.

19.4 Add two additional questions to each of Garvin's quality dimensions presented in Section 19.2.1.

19.5 McCall's quality factors were developed during the 1970s. Almost every aspect of computing has changed dramatically since the time that they were developed, and yet, McCall's factors continue to apply to modern software. Can you draw any conclusions based on this fact?

19.6 Using the subattributes noted for the ISO 9126 quality factor "maintainability" in Section 19.2.3, develop a set of questions that explore whether or not these attributes are present. Follow the example shown in Section 19.2.4.

19.7 Describe the software quality dilemma in your own words.

19.8 What is "good enough" software? Name a specific company and specific products that you believe were developed using the good enough philosophy.

19.9 Considering each of the four aspects of the cost of quality, which do you think is the most expensive and why?

19.10 Do a Web search and find three other examples of "risks" to the public that can be directly traced to poor software quality. Consider beginning your search at **http://catless.ncl.ac.uk/risks**.

19.11 Are *quality* and *security* the same thing? Explain.

19.12 Explain why it is that many of us continue to live by Meskimen's law. What is it about the software business that causes this?

Further Readings and Information Sources

Basic software quality concepts are considered in books by Chemutri (*Master Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*, Ross Publishing, 2010), Henry and Hanlon (*Software Quality Assurance,* Prentice Hall, 2008), Khan and his colleagues (*Software Quality: Concepts and Practice,* Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality,* Springer, 2002), and Daughtrey (*Fundamental Concepts for the Software Quality Engineer,* ASQ Quality Press, 2001).

Duvall and his colleagues (*Continuous Integration: Improving Software Quality and Reducing Risk,* Addison-Wesley, 2007), Tian (*Software Quality Engineering,* Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices,* Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice,* Alpha Science International, Ltd., 2004), and Galin (*Software Quality Assurance: From Theory to Implementation,* Addison-Wesley, 2003) present detailed treatments of SQA. Quality assurance in the context of the agile process is considered by Sterling (*Managing Software Debt, Addison-Wesley, 2010*) and Stamelos and Sfetsos (*Agile Software Development Quality Assurance,* IGI Global, 2007).

Solid design leads to high software quality. Jayasawal and Patton (*Design for Trustworthy Software,* Prentice Hall, 2006) and Ploesch (*Contracts, Scenarios and Prototypes,* Springer, 2004) discuss tools and techniques for developing "robust" software.

Measurement is an important component of software quality engineering. Jones and Bonsignour (*The Economic of Software Quality,* Addison-Wesley, 2011), Ejiogu (*Software Metrics: The Discipline of Software Quality,* BookSurge Publishing, 2005), Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2002), and Nance and Arthur (*Managing Software Quality,* Springer, 2002) discuss important quality-related metrics and models. The team-oriented aspects of software quality are considered by Evans (*Achieving Software Quality through Teamwork,* Artech House Publishers, 2004).

A wide variety of information sources on software quality is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

20 Review Techniques

Key Concepts

bugs. 432
cost-effectiveness . 436
defect amplification. 433
defects. 432
error density. 435
errors. 432
informal reviews. . 439
record keeping . . . 442
review reporting. . 442
sample-driven reviews 444
technical reviews . 441

Software reviews are a "filter" for the software process. That is, reviews are applied at various points during software engineering and serve to uncover errors and defects that can then be removed. Software reviews "purify" software engineering work products, including requirements and design models, code, and testing data. Freedman and Weinberg [Fre90] discuss the need for reviews this way:

> Technical work needs reviewing for the same reason that pencils need erasers: *To err is human.* The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns:
>
> *O wad some power the giftie give us*
> *to see ourselves as other see us*
>
> A review—any review—is a way of using the diversity of a group of people to:
>
> 1. Point out needed improvements in the product of a single person or team;
> 2. Confirm those parts of a product in which improvement is either not desired or not needed;

Quick Look

What is it? You'll make mistakes as you develop software engineering work products. There's no shame in that—as long as you try hard, very hard, to find and correct the mistakes before they are delivered to end users. Technical reviews are the most effective mechanism for finding mistakes early in the software process.

Who does it? Software engineers perform technical reviews, also called peer reviews, with their colleagues.

Why is it important? If you find an error early in the process, it is less expensive to correct. In addition, errors have a way of amplifying as the process proceeds. So a relatively minor error left untreated early in the process can be amplified into a major set of errors later in the project. Finally, reviews save time by reducing the amount of rework that will be required late in the project.

What are the steps? Your approach to reviews will vary depending on the degree of formality you select. In general, six steps are employed, although not all are used for every type of review: planning, preparation, structuring the meeting, noting errors, making corrections (done outside the review), and verifying that corrections have been performed properly.

What is the work product? The output of a review is a list of issues and/or errors that have been uncovered. In addition, the technical status of the work product is also indicated.

How do I ensure that I've done it right? First, select the type of review that is appropriate for your development culture. Follow the guidelines that lead to successful reviews. If the reviews that you conduct lead to higher quality software, you've done it right.

ADVICE

Reviews are like a filter in the software process workflow. Too few, and the flow is "dirty." Too many, and the flow slows to a trickle. Use metrics to determine which reviews work and emphasize them. Remove ineffective reviews from the flow to accelerate the process.

3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software architecture to an audience of customers, management, and technical staff is also a form of review. In this book, however, we focus on *technical or peer reviews*, exemplified by *casual reviews*, *walkthroughs*, and *inspections*. A technical review (TR) is the most effective filter from a quality control standpoint. Conducted by software engineers (and others) for software engineers, the TR is an effective means for uncovering errors and improving software quality.

20.1 Cost Impact of Software Defects

Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end users (or to another framework activity in the software process). In earlier chapters, we used the term *error* to depict a quality problem that is discovered by software engineers (or others) *before* the software is released to the end user (or to another framework activity in the software process).

INFO

Bugs, Errors, and Defects

The goal of software quality control, and in a broader sense, quality management in general, is to remove quality problems in the software. These problems are referred to by various names—*bugs, faults, errors,* or *defects* to name a few. Are each of these terms synonymous, or are there subtle differences between them?

In this book we make a clear distinction between an *error* (a quality problem found *before* the software is released to end users) and a *defect* (a quality problem found only *after* the software has been released to end users).[1] We make this distinction because errors and defects have very different economic, business, psychological, and human impact. As software engineers, we want to find and correct as many errors as possible before the customer and/or end user encounter them. We want to avoid defects—because defects (justifiably) make software people look bad.

It is important to note, however, that the temporal distinction made between errors and defects in this book is *not* mainstream thinking. The general consensus within the software engineering community is that defects and errors, faults, and bugs are synonymous. That is, the point in time that the problem was encountered has no bearing on the term used to describe the problem. Part of the argument in favor of this view is that it is sometimes difficult to make a clear distinction between pre- and postrelease (e.g., consider an incremental process used in agile development).

Regardless of how you choose to interpret these terms, recognize that the point in time at which a problem is discovered does matter and that software engineers should try hard—*very* hard—to find problems before their customers and end users encounter them. If you have further interest in this issue, a reasonably thorough discussion of the terminology surrounding "bugs" can be found at **www.softwaredevelopment.ca/bugs.shtml**.

1 If software process improvement is considered, a quality problem that is propagated from one process framework activity (e.g., **modeling**) to another (e.g., **construction**) can also be called a "defect" (because the problem should have been found before a work product (e.g., a design model) was "released" to the next activity.

ADVICE

The primary objective of an FTR is to find errors before they are passed on to another software engineering activity or released to the end user.

The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, review techniques have been shown to be up to 75 percent effective [Jon86] in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent activities in the software process.

20.2 Defect Amplification and Removal

Quote:

"Some maladies, as doctors say, at their beginning are easy to cure but difficult to recognize . . . but in the course of time when they have not at first been recognized and treated, become easy to recognize but difficult to cure."

Niccolo Machiavelli

A *defect amplification model* [IBM81] can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process. The model is illustrated schematically in Figure 20.1. A box represents a software engineering action. During the action, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

Figure 20.2 illustrates a hypothetical example of defect amplification for a software process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field. Figure 20.3 considers the same conditions except that design and code reviews are conducted as part of each software engineering action. In this case, 10 initial preliminary (architectural) design errors are amplified to 24 errors before testing commences. Only three latent errors

Figure 20.1

Defect amplification model

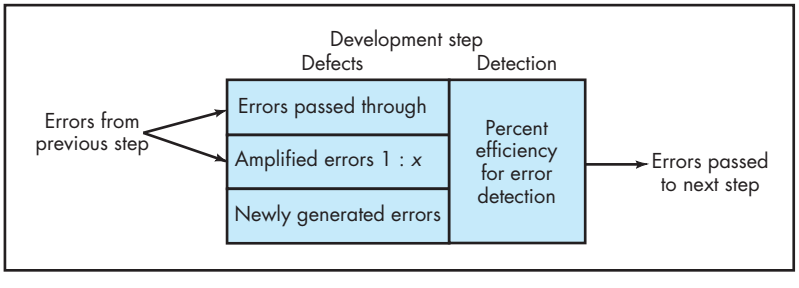


FIGURE 20.2

Defect amplification—no reviews

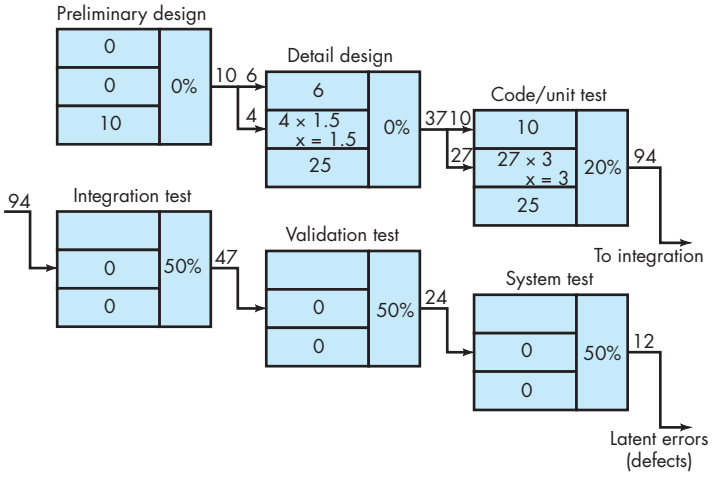
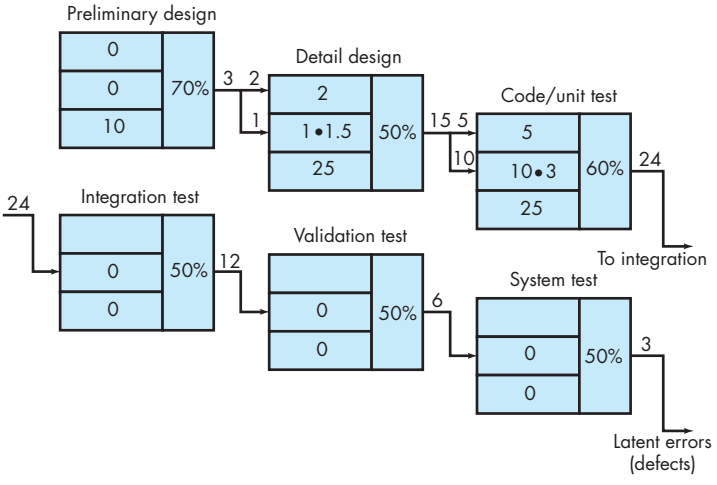


FIGURE 20.3

Defect amplification—reviews conducted



exist. The relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 20.2 and 20.3 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).[2] Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

2 These multipliers are somewhat different from the data presented in Figure 19.2, which is more current. However, they serve to illustrate the costs of defect amplification nicely.

To conduct reviews, you must expend time and effort, and your development organization must spend money. However, the results of the preceding example leave little doubt that you can pay now or pay much more later.

20.3 Review Metrics and Their Use

Technical reviews are one of many actions that are required as part of good software engineering practice. Each action requires dedicated human effort. Since available project effort is finite, it is important for a software engineering organization to understand the effectiveness of each action by defining a set of metrics (Chapter 30) that can be used to assess their efficacy.

Although many metrics can be defined for technical reviews, a relatively small subset can provide useful insight. The following review metrics can be collected for each review that is conducted:

- *Preparation effort,* E_p—the effort (in person-hours) required to review a work product prior to the actual review meeting
- *Assessment effort,* E_a— the effort (in person-hours) that is expended during the actual review
- *Rework effort,* E_r— the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- *Work product size, WPS*—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- *Minor errors found,* Err_{minor}—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- *Major errors found,* Err_{major}—the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

These metrics can be further refined by associating the type of work product that was reviewed for the metrics collected.

20.3.1 Analyzing Metrics

Before analysis can begin, a few simple computations must occur. The total review effort and the total number of errors discovered are defined as:

$$E_{\text{review}} = E_p + E_a + E_r$$

$$\text{Err}_{\text{tot}} = \text{Err}_{\text{minor}} + \text{Err}_{\text{major}}$$

Error density represents the errors found per unit of work product reviewed.

$$\text{Error density} = \frac{\text{Err}_{\text{tot}}}{\text{WPS}}$$

For example, if a requirements model is reviewed to uncover errors, inconsistencies, and omissions, it would be possible to compute the error density in a number of different ways. The requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials. The review uncovers 18 minor errors and 4 major errors. Therefore, $Err_{tot} = 22$. Error density is 1.2 errors per UML diagram or 0.68 errors per requirements model page.

If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases), the percentage of errors uncovered for each review can be computed against the total number of errors found for all reviews. In addition, the error density for each work product can be computed.

Once data are collected for many reviews conducted across many projects, average values for error density enable you to estimate the number of errors to be found in a new (as yet unreviewed document). For example, if the average error density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long, a rough estimate suggests that your software team will find about 19 or 20 errors during the review of the document. If you find only 6 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.

Once testing has been conducted (Chapters 22 through 26), it is possible to collect additional error data, including the effort required to find and correct errors uncovered during testing and the error density of the software. The costs associated with finding and correcting an error during testing can be compared to those for reviews. This is discussed in Section 20.3.2.

20.3.2 Cost-Effectiveness of Reviews

It is difficult to measure the cost-effectiveness of any technical review in real time. A software engineering organization can assess the effectiveness of reviews and their cost benefit only after reviews have been completed, review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured (via testing).

Returning to the example presented in Section 20.3.1, the average error density for requirements models was determined to be 0.6 per page. The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours. The effort required for a major requirement error was found to be 18 person-hours. Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.

Requirements-related errors uncovered during testing require an average of 45 person-hours to find and correct (no data are available on the relative severity of the error). Using the averages noted, we get:

$$\text{Effort saved per error} = E_{testing} - E_{reviews}$$
$$45 - 6 = 30 \text{ person-hours/error}$$

Since 22 errors were found during the review of the requirements model, a saving of about 660 person-hours of testing effort would be achieved. And that's just for requirements-related errors. Errors associated with design and code would add to the overall benefit. The bottom line—effort saved leads to shorter delivery cycles and improved time to market.

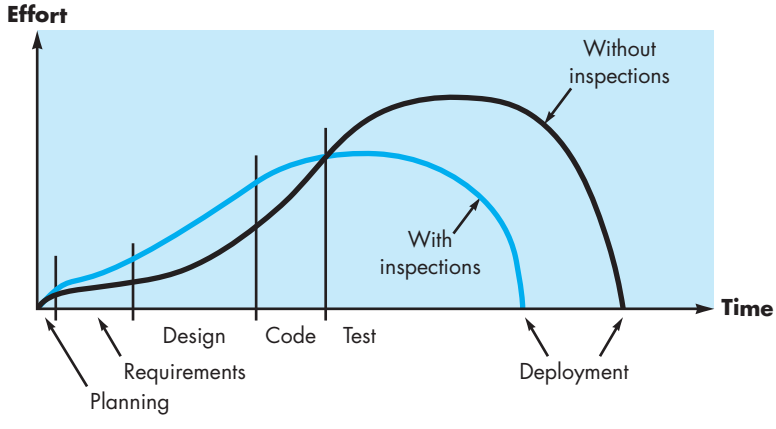
In his book on peer reviews, Karl Wiegers [Wie02] discusses anecdotal data from major companies that have used *inspections* (a relatively formal type of technical review) as part of their software quality control activities. Hewlett-Packard (HP) reported a 10 to 1 return on investment for inspections and noted that actual product delivery accelerated by an average of 1.8 calendar months. AT&T indicated that inspections reduced the overall cost of software errors by a factor of 10 and that quality improved by an order of magnitude and productivity increased by 14 percent. Others report similar benefits. Technical reviews (for design and other technical activities) provide a demonstrable cost benefit and actually save time.

But for many software people, this statement is counterintuitive. "Reviews take time," software people argue, "and we don't have the time to spare!" They argue that time is a precious commodity on every software project and the ability to review "every work product in detail" absorbs too much time.

The examples presented previously in this section indicate otherwise. More importantly, industry data for software reviews has been collected for more than two decades and is summarized qualitatively using the graphs illustrated in Figure 20.4.

FIGURE 20.4

Effort expended with and without reviews
Source: Adapted from [Fag86].



Referring to the figure, the effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced. As important, the deployment date for development with reviews is sooner than the deployment date without reviews. Reviews don't take time, they save it!

20.4 Reviews: A Formality Spectrum

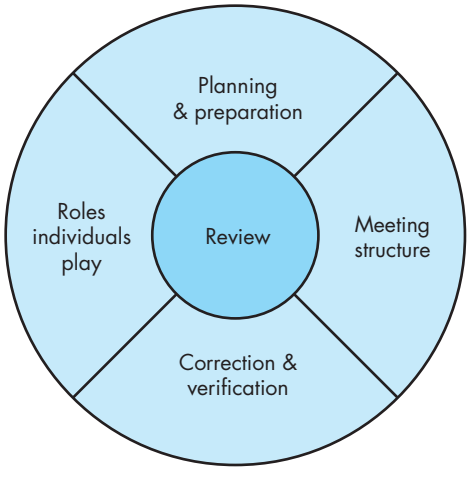
Technical reviews should be applied with a level of formality that is appropriate for the product to be built, the project time line, and the people who are doing the work. Figure 20.5 depicts a reference model for technical reviews [Lai02] that identifies four characteristics that contribute to the formality with which a review is conducted.

Each of the reference model characteristics helps to define the level of review formality. The formality of a review increases when (1) distinct roles are explicitly defined for the reviewers, (2) there is a sufficient amount of planning and preparation for the review, (3) a distinct structure for the review (including tasks and internal work products) is defined, and (4) follow-up by the reviewers occurs for any corrections that are made.

To understand the reference model, let's assume that you've decided to review the interface design for **SafeHomeAssured.com**. You can do this in a variety of different ways that range from relatively casual to extremely rigorous. If you decide that the casual approach is most appropriate, you ask a few colleagues (peers) to examine the interface prototype in an effort to uncover potential problems. All of you decide that there will be no advance preparation, but that you will evaluate the prototype in a reasonably structured way—looking at layout first, aesthetics next, navigation options after that, and so on. As the designer, you decide to take a few notes, but nothing formal.

Figure 20.5

Reference model for technical reviews



But what if the interface is pivotal to the success of the entire project? What if human lives depended on an interface that was ergonomically sound? You might decide that a more rigorous approach was necessary. A review team would be formed. Each person on the team would have a specific role to play—leading the team, recording findings, presenting the material, and so on. Each reviewer would be given access to the work product (in this case, the interface prototype) before the review and would spend time looking for errors, inconsistencies, and omissions. A set of specific tasks would be conducted based on an agenda that was developed before the review occurred. The results of the review would be formally recorded, and the team would decide on the status of the work product based on the outcome of the review. Members of the review team might also verify that the corrections made were done properly.

In this book we consider two broad categories of technical reviews: informal reviews and more formal technical reviews. Within each broad category, a number of different approaches can be chosen. These are presented in the sections that follow.

20.5 Informal Reviews

Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product, or the review-oriented aspects of pair programming (Chapter 5).

A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

One way to improve the efficacy of a desk check review is to develop a set of simple review checklists for each major work product produced by the software team. The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product. For example, let's reexamine a desk check of the interface prototype for **SafeHomeAssured.com**. Rather than simply playing with the prototype at the designer's workstation, the designer and a colleague examine the prototype using a checklist for interfaces:

- Is the layout designed using standard conventions? Left to right? Top to bottom?
- Does the presentation need to be scrolled?
- Are color and placement, typeface, and size used effectively?

- Are all navigation options or functions represented at the same level of abstraction?
- Are all navigation choices clearly labeled?

and so on. Any errors or issues noted by the reviewers are recorded by the designer for resolution at a later time. Desk checks may be scheduled in an ad hoc manner, or they may be mandated as part of good software engineering practice. In general, the amount of material to be reviewed is relatively small and the overall time spent on a desk check span little more than one or two hours.

In Chapter 5, we described *pair programming* in the following manner: XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance.

Pair programming can be characterized as a continuous desk check. Rather than scheduling a review at some point in time, pair programming encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality as a consequence.

In their discussion of the efficacy of pair programming, Williams and Kessler [Wil00] state:

> Anecdotal and initial statistical evidence indicates that pair programming is a powerful technique for productively generating high quality software products. The pair works and shares ideas together to tackle the complexities of software development. They continuously perform inspections on each other's artifacts leading to the earliest, most efficient form of defect removal possible. In addition, they keep each other intently focused on the task at hand.

Some software engineers argue that the inherent redundancy built into pair programming is wasteful of resources. After all, why assign two people to a job that one person can accomplish? The answer to this question can be found in Section 20.3.2. If the quality of the work product produced as a consequence of pair programming is significantly better than the work of an individual, the quality-related savings can more than justify the "redundancy" implied by pair programming.

INFO

Review Checklists

Even when reviews are well organized and properly conducted, it's not a bad idea to provide reviewers with a "crib sheet." That is, it's worthwhile to have a checklist that provides each reviewer with the questions that should be asked about the specific work product that is undergoing review.

One of the most comprehensive collections of review checklists has been developed by NASA at the Goddard Space Flight Center and is available at **http://www.hq.nasa.gov/office/codeq/software/ComplexElectronics/checklists.htm**

Other useful technical review checklists have also been proposed by:

Process Impact

www.processimpact.com/pr_goodies.shtml

Macadamian

www.macadamian.com

20.6 Formal Technical Reviews

A *formal technical review* (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are: (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

Quote:

"There is no urge so great as for one man to edit another man's work."

Mark Twain

The FTR is actually a class of reviews that includes *walkthroughs* and *inspections*. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs, see [Rad02], [Wie02], or [Fre90].

20.6.1 The Review Meeting

WebRef

The NASA SATC *Formal Inspection Guidebook* can be downloaded from **http://www.everyspec.com/NASA/NASA-General/NASA-GB-A302_2418/.**

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing the focus, the FTR has a higher likelihood of uncovering errors.

KEY POINT

An FTR focuses on a relatively small portion of a work product.

The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component). The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required. The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming

familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

ADVICE

In some situations, it's a good idea to have someone other than the producer walk through the product undergoing review. This leads to a literal interpretation of the work product and better error recognition.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of a *recorder,* that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each.

At the end of the review, all attendees of the FTR must decide whether to: (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

20.6.2 Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting, and a *review issues list* is produced. In addition, a *formal technical review summary report* is completed. A review summary report answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single-page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The review issues list serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow-up to the review leader.

20.6.3 Review Guidelines

Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review

that is uncontrolled can often be worse than no review at all. The following represents a minimum set of guidelines for formal technical reviews:

ADVICE

Don't point out errors harshly. One way to be gentle is to ask a question that enables the producer to discover the error.

1. *Review the product, not the producer.* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.
2. *Set an agenda and maintain it.* One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.

Quote:

"A meeting is too often an event in which minutes are taken and hours are wasted."

Author unknown

3. *Limit debate and rebuttal.* When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. *Enunciate problem areas, but don't attempt to solve every problem noted.* A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. *Take written notes.* It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded. Alternatively, notes may be entered directly into a notebook computer.
6. *Limit the number of participants and insist upon advance preparation.* Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).
7. *Develop a checklist for each product that is likely to be reviewed.* A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even testing work products.
8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as a task during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.

Quote:

"It is one of the most beautiful compensations of life, that no man can sincerely try to help another without helping himself."

Ralph Waldo Emerson

9. *Conduct meaningful training for all reviewers.* To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg [Fre90] estimate a one-month learning curve for every 20 people who are to participate effectively in reviews.
10. *Review your early reviews.* Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a software organization should experiment to determine what approach works best in a local context.

20.6.4 Sample-Driven Reviews

In an ideal setting, every software-engineering work product would undergo a formal technical review. In the real word of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized.

Thelin and his colleagues [The01] suggest a sample-driven review process in which samples of all software engineering work products are inspected to determine which work products are most error prone. Full FTR resources are then focused only on those work products that are likely (based on data collected during sampling) to be error prone.

ADVICE

Reviews take time, but it's time well spent. However, if time is short and you have no other option, do not dispense with reviews. Rather, use sample-driven reviews.

To be effective, the sample-driven review process must attempt to quantify those work products that are primary targets for full FTRs. To accomplish this, the following steps are suggested [The01]:

1. Inspect a fraction a_i of each software work product i. Record the number of faults f_i found within a_i.
2. Develop a gross estimate of the number of faults within work product i by multiplying f_i by $1/a_i$.
3. Sort the work products in descending order according to the gross estimate of the number of faults in each.
4. Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must be representative of the work product as a whole and large enough to be meaningful to the reviewers who do the sampling. As a_i increases, the likelihood that the sample is a valid representation of the work product also increases. However, the resources required to

do sampling also increase. A software engineering team must establish the best value for a_i for the types of work products produced.[3]

SafeHome

Quality Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and other members of the product software engineering team.

The conversation:

Doug: I know we didn't spend time developing a quality plan for this project, but we're already into it and we have to consider quality . . . right?

Jamie: Sure. We've already decided that as we develop the requirements model [Chapters 9–11], Ed has committed to develop a testing procedure for each requirement.

Doug: That's really good, but we're not going to wait until testing to evaluate quality, are we?

Vinod: No! Of course not. We've got reviews scheduled into the project plan for this software increment. We'll begin quality control with the reviews.

Jamie: I'm a bit concerned that we won't have enough time to conduct all the reviews. In fact, I know we won't.

Doug: Hmmm. So what do you propose?

Jamie: I say we select those elements of the requirements and design model that are most critical to *SafeHome* and review them.

Vinod: But what if we miss something in a part of the model we don't review?

Shakira: I read something about a sampling technique [Section 20.6.4] that might help us target candidates for review. (Shakira explains the approach.)

Jamie: Maybe . . . but I'm not sure we even have time to sample every element of the models.

Vinod: What do you want us to do, Doug?

Doug: Let's steal something from Extreme Programming [Chapter 5]. We'll develop the elements of each model in pairs—two people—and conduct an informal review of each as we go. We'll then target "critical" elements for a more formal team review, but keep those reviews to a minimum. That way, everything gets looked at by more than one set of eyes, but we still maintain our delivery dates.

Jamie: That means we're going to have to revise the schedule.

Doug: So be it. Quality trumps schedule on this project.

20.7 Post-Mortem Evaluations

Many lessons can be learned if a software team takes the time to evaluate the results of a software project after the software has been delivered to end users. Baaz and his colleagues [Baa10] suggest the use of a *post-mortem evaluation* (PME) as a mechanism to determine what went right and what went wrong when software engineering process and practice are applied in a specific project.

Unlike an FTR that focuses on a specific work product, a PME examines the entire software project, focusing on both "*excellences* (that is, achievements

3 Thelin and his colleagues have conducted a detailed simulation that can assist in making this determination. See [The01] for details.

and positive experiences) and *challenges* (problems and negative experiences)" [Baa10]. Often conducted in a workshop format, a PME is attended by members of the software team and stakeholders. The intent is to identify excellences and challenges and to extract lessons learned from both. The objective is to suggest improvements to both process and practice going forward.

20.8 Summary

The intent of every technical review is to find errors and uncover issues that would have a negative impact on the software to be deployed. The sooner an error is uncovered and corrected, the less likely that error will propagate to other software engineering work products and amplify itself, resulting in significantly more effort to correct it.

To determine whether quality control activities are working, a set of metrics should be collected. Review metrics focus on the effort required to conduct the review and the types and severity of errors uncovered during the review. Once metrics data are collected, they can be used to assess the efficacy of the reviews you do conduct. Industry data indicates that reviews provide a significant return on investment.

A reference model for review formality identifies the roles people play, planning and preparation, meeting structure, correction approach, and verification as the characteristics that indicate the degree of formality with which a review is conducted. Informal reviews are casual in nature, but can still be used effectively to uncover errors. Formal reviews are more structured and have the highest probability of leading to high-quality software.

Informal reviews are characterized by minimal planning and preparation and little record keeping. Desk checks and pair programming fall into the informal review category.

A formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors. Walkthroughs and inspections establish defined roles for each reviewer, encourage planning and advance preparation, require the application of defined review guidelines, and mandate record keeping and status reporting. Sample-driven reviews can be used when it is not possible to conduct formal technical reviews for all work products.

Problems and Points to Ponder

20.1 Explain the difference between an *error* and a *defect*.

20.2 Why can't we just wait until testing to find and correct all software errors?

20.3 Assume that 10 errors have been introduced in the requirements model and that each error will be amplified by a factor of 2:1 into design and an addition 20 design errors are introduced and then amplified 1.5:1 into code where an additional 30 errors are introduced.

Assume further that all unit testing will find 30 percent of all errors, integration will find 30 percent of the remaining errors, and validation tests will find 50 percent of the remaining errors. No reviews are conducted. How many errors will be released to the field.

20.4 Reconsider the situation described in Problem 20.3, but now assume that requirements, design, and code reviews are conducted and are 60 percent effective in uncovering all errors at that step. How many errors will be released to the field?

20.5 Reconsider the situation described in Problems 20.3 and 20.4. If each of the errors released to the field costs $4,800 to find and correct and each error found in review costs $240 to find and correct, how much money is saved by conducting reviews?

20.6 Describe the meaning of Figure 20.4 in your own words.

20.7 Which of the reference model characteristics do you think has the strongest bearing on review formality? Explain why.

20.8 Can you think of a few instances in which a desk check might create problems rather than provide benefits?

20.9 A formal technical review is effective only if everyone has prepared in advance. How do you recognize a review participant who has not prepared? What do you do if you're the review leader?

20.10 Considering all of the review guidelines presented in Section 20.6.3, which do you think is most important and why?

Further Readings and Information Sources

Although there have been relatively few new books written on software reviews over the past decade, recent additions to the literature include books by McCann (*Cost-Benefit Analysis of Quality Practices,* IEEE Press, 2012), Wong (*Modern Software Review,* IRM Press, 2006), and Young (*Project Requirements: A Guide to Best Practices,* Management Concepts, 2006). Older contributions that provide worthwhile guidance include: Radice (*High Quality, Low Cost Software Inspections,* Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001) and Gilb and Graham (*Software Inspection,* Addison-Wesley, 1993). Freedman and Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews,* Dorset House, 1990) remains a classic text and continues to provide worthwhile information about this important subject.

Books by Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process,* Addison-Wesley, 2012) and Adkins (*Coaching Agile Teams: A Companions for ScrumMasters, Agile Coaches, and Project Managers in Transition,* Addison-Wesley, 2010) describe the roles of reviews in agile software processes.

A wide variety of information sources on software reviews is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

21 Software Quality Assurance

Key Concepts

elements of software quality assurance . 450
formal approaches. 456
goals 454
ISO 9001:2008 standard. 462
quality management resources 452
Six Sigma. 458
software reliability 459
software safety . . 460
SQA plan 463
SQA tasks 453
statistical software quality assurance . 456

The software engineering approach described in this book works toward a single goal: to produce on-time, high-quality software. Yet many readers will be challenged by the question: "What is software quality?"

Philip Crosby [Cro79], in his landmark book on quality, provides a wry answer to this question:

> The problem of quality management is not what people don't know about it. The problem is what they think they do know . . .
>
> In this regard, quality has much in common with sex. Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

Quick Look

What is it? It's not enough to talk the talk by saying that software quality is important. You have to (1) explicitly define what is meant when you say "software quality," (2) create a set of activities that will help ensure that every software engineering work product exhibits high quality, (3) perform quality control and assurance activities on every software project, (4) use metrics to develop strategies for improving your software process and, as a consequence, the quality of the end product.

Who does it? Everyone involved in the software engineering process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time to market.

What are the steps? Before software quality assurance (SQA) activities can be initiated, it is important to define *software quality* at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

What is the work product? A Software Quality Assurance Plan is created to define a software team's SQA strategy. During modeling and coding, the primary SQA work product is the output of technical reviews (Chapter 20). During testing (Chapters 22 through 26), test plans and procedures are produced. Other work products associated with process improvement may also be generated.

How do I ensure that I've done it right? Find errors before they become defects! That is, work to improve your defect removal efficiency (Chapter 30), thereby reducing the amount of rework that your software team has to perform.

Indeed, quality is a challenging concept—one that we addressed in some detail in Chapter 19.[1]

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! *Software quality assurance* (often called *quality management*) is an umbrella activity (Chapter 3) that is applied throughout the software process.

Software quality assurance (SQA) encompasses: (1) an SQA process, (2) specific quality assurance and quality control tasks (including technical reviews and a multi-tiered testing strategy), (3) effective software engineering practice (methods and tools), (4) control of all software work products and the changes made to them (Chapter 29), (5) a procedure to ensure compliance with software development standards (when applicable), and (6) measurement and reporting mechanisms.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does "the right things at the right time in the right way."

21.1 Background Issues

Quality control and assurance are essential activities for any business that produces products to be used by others. Prior to the twentieth century, quality control was the sole responsibility of the craftsperson who built a product. As time passed and mass production techniques became commonplace, quality control became an activity performed by people other than the ones who built the product.

Quote:

"You made too many wrong mistakes."

Yogi Berra

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement [Dem86] as key elements of quality management.

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE93a]. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of

1 If you have not read Chapter 19, you should do so now.

actions" [Sch98c] that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility — software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors noted in Chapter 19? Have software engineering practices been conducted according to preestablished standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

21.2 Elements of Software Quality Assurance

WebRef

An in-depth discussion of SQA, including a wide array of definitions, can be obtained at **http://www.swqual.com/images/FoodforThought_Jan2011.pdf.**

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner [Hor03]:

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers (Chapter 20). Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing (Chapters 22 through 26) is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to

better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices (Chapter 29) have been instituted.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement (Chapter 37) and is a key proponent and sponsor of educational programs.

Quote:

"Excellence is the unlimited ability to improve the quality of what you have to offer."

Rick Petin

Vendor management. Three categories of software are acquired from external software vendors—*shrink-wrapped packages* (e.g., Microsoft Office), a *tailored shell* [Hor03] that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and *contracted software* that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.

Security management. With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security (Chapter 27).

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. Although the analysis and mitigation of risk (Chapter 35) is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

In addition to each of these concerns and activities, SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern.

INFO

Quality Management Resources

Dozens of quality management resources are available on the Web, including professional societies, standards organizations, and general information sources. The sites that follow provide a good starting point:

American Society for Quality (ASQ) Software Division
www.asq.org/software

Association for Computer Machinery
www.acm.org

Cyber Security and Information Systems Information Analysis Center (CSIAC)
https://sw.thecsiac.com/

International Organization for Standardization (ISO)
www.iso.ch

ISO SPICE **http://www.spiceusergroup.org/**

Malcolm Baldridge National Quality Award
http://www.nist.gov/baldrige/

Software Engineering Institute **www.sei.cmu.edu/**

Software Testing and Quality Engineering
www.stickyminds.com

Six Sigma Resources **www.isixsigma.com/**
www.asq.org/sixsigma/

TickIT International: Quality certification topics
www.tickit.org/international.htm

Total Quality Management (TQM)
http://www.isixsigma.com/methodology/total-quality-management-tqm/
http://asq.org/learn-about-quality/total-quality-management/overview/overview.html

21.3 SQA Processes and Product Characteristics

As we begin a discussion of software quality assurance, it's important to note that SQA procedures and approaches that work in one software environment may not work as well in another. Even within a company that adopts a consistent approach[2] to software engineering, different software products may exhibit different levels of quality [Par11].

The solution to this dilemma is to understand the specific quality requirements for a software product and then select the process and specific SQA actions and tasks that will be used to meet those requirements. The Software Engineering Institute's CMMI and ISO 9000 standards are the most commonly used process frameworks. Each proposes "a syntax and semantics" [Par11] that will lead to the implementation of software engineering practices that improve product quality. Rather than instantiating either framework in its entirety, a software organization can "harmonize" the two models by selecting elements of both frameworks and matching them to the quality requirements of an individual product.

21.4 SQA Tasks, Goals, and Metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and

2 For example, CMMI-defined process and practices (Chapter 37).

an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

21.4.1 SQA Tasks

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

? What is the role of an SQA group?

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

Quote:

"Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives."

William A. Foster

In addition to these activities, the SQA group coordinates the control and management of change (Chapter 29) and helps to collect and analyze software metrics.

SafeHome

Software Quality Assurance

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and other members of the product software engineering team.

The conversation:

Doug: How are things going with the informal reviews?

Jamie: We're conducting informal reviews of the critical project elements in pairs as we code but before testing. It's going faster than I thought.

Doug: That's good, but I want to have Bridget Thorton's SQA group conduct audits of our work products to ensure that we're following our processes and meeting our quality goals.

Venod: Aren't they already doing the bulk of the testing?

Doug: Yes, they are. But QA is more than testing. We need to be sure that our documents are evolving along with our code and that we're making sure we don't introduce errors as we integrate new components.

Jamie: I really don't want to be evaluated based on their findings.

Doug: No worries. The audits are focused on conformance of our work products to the requirements and process our activities. We'll only be using audit results to try to improve our processes as well as our software products.

Vinod: I have to believe it's going to take more of our time.

Doug: In the long run it will save us time when we find defects earlier. It also costs less to fix defects if they're caught early.

Jamie: That sounds like a good thing then.

Doug: It's also important to identify the activities where defects were introduced and add review tasks to catch them in the future.

Vinod: That'll help us determine if we're sampling carefully enough with our review activities.

Doug: I think SQA activities will make us a better team in the long run.

21.4.2 Goals, Attributes, and Metrics

The SQA activities described in the preceding section are performed to achieve a set of pragmatic goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality. Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality

FIGURE 21.1 **Software quality goals, attributes, and metrics**

Source: Adapted from [Hya96].

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Figure 21.1 (adapted from [Hya96]) identifies the attributes that are indicators for the existence of quality for each of the goals discussed. Metrics that can be used to indicate the relative strength of an attribute are also shown.

21.5 Formal Approaches to SQA

In the preceding sections, we have argued that software quality is everyone's job and that it can be achieved through competent software engineering practice as well as through the application of technical reviews, a multi-tiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards and process frameworks. In addition, quality can be defined in terms of a broad array of quality attributes and measured (indirectly) using a variety of indices and metrics.

Over the past three decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements (Chapter 28) is available. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

Attempts to prove programs correct are not new. Dijkstra [Dij76a] and Linger, Mills, and Witt [Lin79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 14).

21.6 Statistical Software Quality Assurance

Quote:

"20 percent of the code has 80 percent of the errors. Find them, fix them!"

Lowell Arthur

Statistical quality assurance reflects a growing trend throughout the industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

? What steps are required to perform statistical SQA?

This relatively simple concept represents an important step toward the creation of an adaptive software process in which changes are made to improve those elements of the process that introduce error.

21.6.1 A Generic Example

Quote:

"A statistical analysis, properly conducted, is a delicate dissection of uncertainties, a surgery of suppositions."

M. J. Moroney

To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on errors and defects for a period of one year. Some of the errors are uncovered as software is being developed. Other defects are encountered after the software has been released to its end users. Although hundreds of different problems are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or erroneous specifications (IES).
- Misinterpretation of customer communication (MCC).
- Intentional deviation from specifications (IDS).
- Violation of programming standards (VPS).
- Error in data representation (EDR).
- Inconsistent component interface (ICI).
- Error in design logic (EDL).
- Incomplete or erroneous testing (IET).
- Inaccurate or incomplete documentation (IID).
- Error in programming language translation of design (PLT).
- Ambiguous or inconsistent human/computer interface (HCI).
- Miscellaneous (MIS).

To apply statistical SQA, the table in Figure 21.2 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once

FIGURE 21.2

Data collection for statistical SQA

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, you might implement requirements gathering techniques (Chapter 8) to improve the quality of customer communication and specifications. To improve EDR, you might acquire tools for data modeling and perform more stringent data design reviews.

It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement (e.g., [Rya11], [Art97]). In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

21.6.2 Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects in manufacturing and service-related processes" [ISI08]. The term *Six Sigma* is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high-quality standard. The Six Sigma methodology defines three core steps:

? What are the core steps of the Six Sigma methodology?

- *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication.
- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

- *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements.

- *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject. If you have further interest, see [ISI08], [Pyz03], and [Sne03].

21.7 Software Reliability

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

Software reliability, unlike many other quality factors, can be measured directly and estimated using historical and developmental data. *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [Mus87]. To illustrate, program X is estimated to have a reliability of 0.999 over eight elapsed processing hours. In other words, if program X were to be executed 1000 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 999 times.

Quote:

"The unavoidable price of reliability is simplicity."

C. A. R. Hoare

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term *failure*? In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds, while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

KEY POINT

Software reliability problems can almost always be traced to defects in design or implementation.

21.7.1 Measures of Reliability and Availability

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear (see Chapter 1) does not enter into the picture.

There has been an ongoing debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet to be established, it is worthwhile to consider a few simple concepts that apply to both system elements.

KEY POINT

It is important to note that MTBF and related measures are based on CPU time, not wall clock time.

If we consider a computer-based system, a simple measure of reliability is *mean-time-between-failure* (MTBF):

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

where the acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair*,[3] respectively.

Many researchers argue that MTBF is a far more useful measure than other quality-related software metrics discussed in Chapter 30. Stated simply, an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 3000 processor hours without failure. Many defects in this program may remain undetected for tens of thousands of hours before they are discovered. The MTBF of such obscure errors might be 30,000 or even 60,000 processor hours. Other defects, as yet undiscovered, might have a failure rate of 4000 or 5000 hours. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

However, MTBF can be problematic for two reasons: (1) it projects a time span between failures, but does not provide us with a projected failure rate, and (2) MTBF can be misinterpreted to mean average life span even though this is *not* what it implies.

An alternative measure of reliability is *failures-in-time* (FIT)—a statistical measure of how many failures a component will have over 1 billion hours of operation. Therefore, 1 FIT is equivalent to one failure in every billion hours of operation.

ADVICE

Some aspects of availability (not discussed here) have nothing to do with failure. For example, scheduling downtime (for support functions) causes the software to be unavailable.

In addition to a reliability measure, you should also develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})} \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software. For a comprehensive discussion of software reliability measures, see [Laz11].

Quote:

"The safety of the people shall be the highest law."

Cicero

21.7.2 Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

3 Although debugging (and related corrections) may be required as a consequence of failure, in many cases the software will work properly after a restart with no other change.

Quote:

"I cannot imagine any condition which would cause this ship to founder. Modern shipbuilding has gone beyond that."

E. I. Smith, captain of the *Titanic*

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be: (1) causes uncontrolled acceleration that cannot be stopped, (2) does not respond to depression of brake pedal (by turning off), (3) does not engage when switch is activated, and (4) slowly loses or gains speed. Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.[4] To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If and only if a set of external environmental conditions is met, the improper position of the mechanical device will cause a disastrous failure. Analysis techniques [Eri05] such as fault tree analysis, real-time logic, and Petri net models can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

WebRef

A worthwhile collection of papers on software safety can be found at **www.safeware-eng.com/.**

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system and its environment.

A comprehensive discussion of software safety is beyond the scope of this book. If you have further interest in software safety and related system issues, see [Fir12], [Har12], [Smi05], and [Lev95].

21.8 The ISO 9000 Quality Standards[5]

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management [ANS87]. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's

4 This approach is similar to the risk analysis methods described in Chapter 35. The primary difference is the emphasis on technology issues rather than project-related topics.

5 This section, written by Michael Stovsky, has been adapted from *Fundamentals of ISO 9000*, a workbook developed for *Essential Software Engineering*, a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard.

WebRef

Extensive links to ISO 9000/9001 resources can be found at **www.tantara.ab.ca/info.htm.**

The requirements delineated by ISO 9001:2008 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001:2008, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. If you desire further information on ISO 9001:2008, see [Coc11], [Hoy09], or [Cia09].

INFO

The ISO 9001:2008 Standard

The following outline defines the basic elements of the ISO 9001:2000 standard. Comprehensive information on the standard can be obtained from the International Organization for Standardization (**www.iso.ch**) and other Internet sources (e.g., **www.praxiom.com**).

Establish the elements of a quality management system.
- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system.
- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.
- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction.
- Define a quality plan that addresses objectives, responsibilities, and authority.
- Define communication mechanisms among stakeholders.

Establish review mechanisms for the quality management system.
- Identify review methods and feedback mechanisms.
- Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

Establish control mechanisms.
- For planning.
- For customer requirements.
- For technical activities (e.g., analysis, design, testing).
- For project monitoring and management.

Define methods for remediation.
- Assess quality data and metrics.
- Define approach for continuous process and quality improvement.

21.9 The SQA Plan

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

A standard for SQA plans has been published by the IEEE [IEE93]. The standard recommends a structure that identifies: (1) the purpose and scope of the plan, (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA, (3) all applicable standards and practices that are applied during the software process, (4) SQA actions and tasks (including reviews and audits) and their placement throughout the software process, (5) the tools and methods that support SQA actions and tasks, (6) software configuration management (Chapter 29) procedures, (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and (8) organizational roles and responsibilities relative to product quality.

Software Tools

Software Quality Management

Objective: The objective of SQA tools is to assist a project team in assessing and improving the quality of software work product.

Mechanics: Tools mechanics vary. In general, the intent is to assess the quality of a specific work product. Note: A wide array of software testing tools (see Chapters 22 through 26) are often included within the SQA tools category.

Representative Tools:[6]

QA Complete, developed by SmartBear (**http://smartbear.com/products/qa-tools/test-management**), QA management ensures complete test coverage through every stage of the software development process.

QPR Suite, developed by QPR Software (**http://www.qpr.com**), provides support for Six Sigma and other quality management approaches.

Quality Tools and Templates, developed by iSixSigma (**http://www.isixsigma.com/tools-templates/**), describe a wide array of useful tools and methods for quality management.

NASA Quality Resources, developed by the Goddard Space Flight Center (**http://www.hq.nasa.gov/office/codeq/software/ComplexElectronics/checklists.htm**) provides useful forms, templates, checklists, and tools for SQA.

21.10 Summary

Software quality assurance is a software engineering umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, oversight of quality control activities such as technical reviews and software testing, procedures for change

6 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

management, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

In summary, you should note the words of Dunn and Ullman [Dun82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

Problems and Points to Ponder

21.1. Some people say that "variation control is the heart of quality control." Since every program that is created is different from every other program, what are the variations that we look for and how do we control them?

21.2. Is it possible to assess the quality of software if the customer keeps changing what it is supposed to do?

21.3. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss the differences.

21.4. Can a program be correct and still not be reliable? Explain.

21.5. Can a program be correct and still not exhibit good quality? Explain.

21.6. Why is there often tension between a software engineering group and an independent software quality assurance group? Is this healthy?

21.7. You have been given the responsibility for improving the quality of software across your organization. What is the first thing that you should do? What's next?

21.8. Besides counting errors and defects, are there other countable characteristics of software that imply quality? What are they and can they be measured directly?

21.9. The MTBF concept for software is open to criticism. Explain why.

21.10. Consider two safety-critical systems that are controlled by computer. List at least three hazards for each that can be directly linked to software failures.

21.11. Acquire a copy of ISO 9001:2000 and ISO 9000-3. Prepare a presentation that discusses three ISO 9001 requirements and how they apply in a software context.

Further Readings and Information Sources

Books by Chemuturi (*Mastering Software Quality Assurance,* J. Ross Publishing, 2010), Hoyle (*Quality Management Essentials,* Butterworth-Heinemann, 2007), Tian (*Software Quality Engineering,* Wiley-IEEE Computer Society Press, 2005), El Emam (*The ROI from Software Quality,* Auerbach, 2005), Horch (*Practical Guide to Software Quality Management,* Artech

House, 2003), and Nance and Arthur (*Managing Software Quality,* Springer, 2002) are excellent management-level presentations on the benefits of formal quality assurance programs for computer software. Books by Deming [Dem86], Defoe and Juran (*Juran's Quality Handbook,* 6th ed., McGraw-Hill, 2010), Juran (*Juran on Quality by Design,* Free Press, 1992), and Crosby ([Cro79] and *Quality Is Still Free,* McGraw-Hill, 1995) do not focus on software, but are must reading for senior managers with software development responsibility. Gluckman and Roome (*Everyday Heroes of the Quality Movement,* Dorset House, 1993) humanizes quality issues by telling the story of the players in the quality process. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995) presents a quantitative view of software quality.

Work by Evans (*Quality & Performance Excellence,* South-Western College Publishing, 2007) and (*Total Quality: Management, Organization and Strategy,* 4th ed., South-Western College Publishing, 2004), Bru (*Six Sigma for Managers,* McGraw-Hill, 2005), and Dobb (*ISO 9001:2000 Quality Registration Step-by-Step,* 3rd ed., Butterworth-Heinemann, 2004) are representative of the many books written on TQM, Six Sigma, and ISO 9001:2000, respectively.

O'Connor and Kleyner (*Practical Reliability Engineering,* Wiley, 2012), Naik and Tripathy (*Software Testing and Quality Assurance: Theory and Practice,* Wiley-Spektrum, 2008), Pham (*System Software Reliability*, Springer, 2006), Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* 2nd ed., McGraw-Hill, 2004) and Peled (*Software Reliability Methods*, Springer, 2001) have written practical guides that describe methods for measuring and analyzing software reliability.

Vincoli (*Basic Guide to System Safety,* Wiley, 2006), Dhillon (*Computer System Reliability, Safety and Usability*, CRC Press, 2013) and (*Engineering Safety,* World Scientific Publishing Co., 2003), Hermann (*Software Safety and Reliability,* Wiley-IEEE Computer Society Press, 2010), Verma, Ajit, and Karanki (*Reliability and Safety Engineering*, Springer, 2010), Storey (*Safety-Critical Computer Systems,* Addison-Wesley, 1996), and Leveson [Lev95] are the most comprehensive discussions of software and system safety published to date. In addition, van der Meulen (*Definitions for Hardware and Software Safety Engineers,* Springer-Verlag, 2000) offers a complete compendium of important concepts and terms for reliability and safety; Gardiner (*Testing Safety-Related Software,* Springer-Verlag, 1999) provides specialized guidance for testing safety critical systems; Friedman and Voas (*Software Assessment: Reliability Safety and Testability,* Wiley, 1995) provide useful models for assessing reliability and safety. Ericson (*Hazard Analysis Primer,* CreateSpace Independent Publishing Platform, 2012) and (*Hazard Analysis Techniques for System Safety,* Wiley, 2005) addresses the increasingly important domain of hazard analysis.

A wide variety of information sources on software quality assurance and related topics is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website **www.mhhe.com/pressman**.

CHAPTER

22 Software Testing Strategies

Key Concepts

alpha test 485
beta test 485
bottom-up integration 477
class testing 481
cluster testing 482
completion 472
configuration review 484
debugging 488
deployment testing 487
drivers 475
independent test group 469
integration testing 475
object-oriented software 481
performance testing 487
recovery testing .. 486
regression testing . 478
security testing ... 486
smoke testing 479
stress testing 487
stubs 475
system testing ... 486
test strategies for MobileApps 483
test strategies for WebApps 482

A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test-case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses. Shooman [Sho83] discusses these issues:

> In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defense against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and technical reviews] are helping us to reduce the number of initial errors that are inherent in the code. Similarly, different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

These "approaches and philosophies" are what we call *strategy*—the topic to be presented in this chapter. In Chapters 23 through 26, the testing methods and techniques that implement the strategy are presented.

22.1 A Strategic Approach to Software Testing

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

Quick Look

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed. But how do you conduct the tests? Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you've already conducted as you add new components to a large system? When should you involve the customer? These and many other questions are answered when you develop a software testing strategy.

Who does it? A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering action. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps? Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component or on a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

What is the work product? A *Test Specification* documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the types of tests that will be conducted.

How do I ensure that I've done it right? By reviewing the *Test Specification* prior to testing, you can assess the completeness of test cases and testing tasks. An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

thread-based testing 482
top-down integration 476
unit testing 473
validation 468
validation testing . 483
verification 468

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews (Chapter 20). By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

WebRef
Useful resources for software testing can be found at **www.mtsu.edu/~storm/.**

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

22.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of tasks that ensure that software correctly implements a specific function. *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

> Verification: "Are we building the product right?"
>
> Validation: "Are we building the right product?"

Quote:

"Testing is the unavoidable part of any responsible effort to develop a software system."

William Howden

The definition of V&V encompasses many software quality assurance activities (Chapter 21).[1]

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

ADVICE

Don't get sloppy and view testing as a "safety net" that will catch all errors that occurred because of weak software engineering practices. It won't. Stress quality and error detection throughout the software process.

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller [Mil77] relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

22.1.2 Organizing for Software Testing

Quote:

"Optimism is the occupational hazard of programming; testing is the treatment."

Kent Beck

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error-free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigates against thorough testing.

---

1 It should be noted that there is a strong divergence of opinion about what types of testing constitute "validation." Some people believe that *all* testing is verification and that validation is conducted when requirements are reviewed and approved, and later, by the user when the system is operational. Other people view unit and integration testing (Sections 22.3.1 and 22.3.2) as verification and higher-order testing (Sections 22.6 and 22.7) as validation.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyzes, models, and then creates a computer program and its documentation. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) destructive. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than to uncover errors. Unfortunately, errors will be nevertheless present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that you might infer from the preceding discussion: (1) that the developer of software should do no testing at all, (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly, (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

KEY POINT

An independent test group does not have the "conflict of interest" that builders of the software might experience.

The role of an *independent test group* (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

However, you don't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

Quote:

"The first mistake that people make is thinking that the testing team is responsible for assuring quality."

Brian Marick

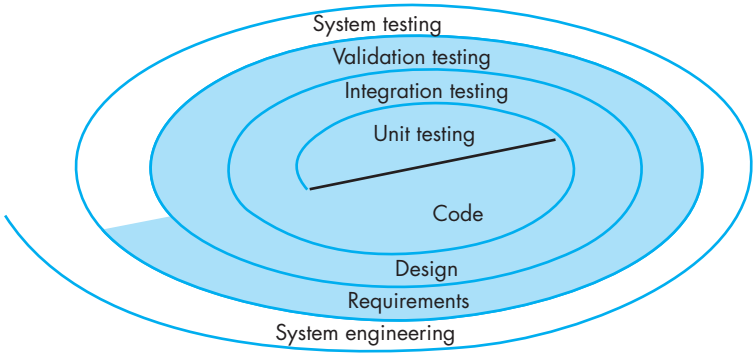
The ITG is part of the software development project team in the sense that it becomes involved during analysis and design and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering team.

22.1.3 Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in Figure 22.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop

FIGURE 22.1

Testing strategy



computer software, you spiral inward along streamlines that decrease the level of abstraction on each turn.

? What is the overall strategy for software testing?

A strategy for software testing may also be viewed in the context of the spiral (Figure 22.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter *validation testing,* where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

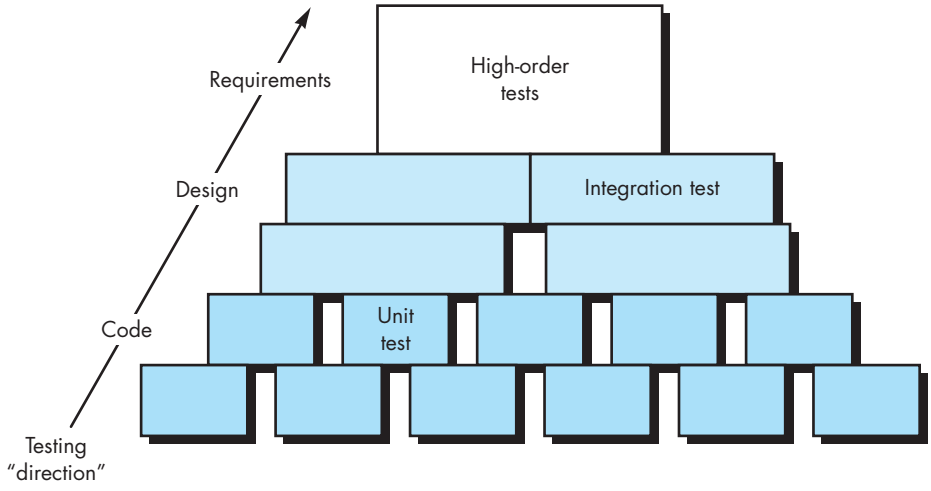
WebRef

Useful resources for software testers can be found at **www.SQAtester.com.**

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 22.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Test-case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all functional, behavioral, and performance requirements.

FIGURE 22.2

Software testing steps



The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

SAFEHOME

Preparing for Testing

The scene: Doug Miller's office, as component-level design continues and construction of certain components begins.

The players: Doug Miller, software engineering manager, Vinod, Jamie, Ed, and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Doug: It seems to me that we haven't spent enough time talking about testing.

Vinod: True, but we've all been just a little busy. And besides, we have been thinking about it . . . in fact, more than thinking.

Doug (smiling): I know . . . we're all overloaded, but we've still got to think down the line.

Shakira: I like the idea of designing unit tests before I begin coding any of my components, so that's what I've been trying to do. I have a pretty big file of tests to run once code for my components is complete.

Doug: That's an Extreme Programming [an agile software development process, see Chapter 5] concept, no?

Ed: It is. Even though we're not using Extreme Programming per se, we decided that it'd be a good idea to design unit tests before we build the component—the design gives us all of the information we need.

Jamie: I've been doing the same thing.

Vinod: And I've taken on the role of the integrator, so every time one of the guys passes a component to me, I'll integrate it and run a series of regression tests on the partially integrated program. I've been working to design a set of appropriate tests for each function in the system.

Doug (to Vinod): How often will you run the tests?

Vinod: Every day . . . until the system is integrated . . .well, I mean until the software increment we plan to deliver is integrated.

Doug: You guys are way ahead of me!

Vinod (laughing): Anticipation is everything in the software biz, Boss.

22.1.4 Criteria for Completion of Testing

? When are we finished testing?

A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Quote:

"Testing only to end-user requirements is like inspecting a building based on the work done by the interior designer at the expense of the foundations, girders, and plumbing."

Boris Beizer

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The *cleanroom software engineering* approach (Chapter 28) suggests statistical use techniques [Kel00] that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. By collecting metrics during software testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"

22.2 Strategic Issues

? What guidelines lead to a successful software testing strategy?

Later in this chapter, we present a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [Gil95] argues that a software testing strategy will succeed only when software testers: (1) specify product requirements in a quantifiable manner long before testing commences, (2) state testing objectives explicitly, (3) understand the users of the software and develop a profile for each user category, (4) develop a testing plan that emphasizes "rapid cycle testing,"[2] (5) build "robust" software that is designed to test itself (the concept of antibugging is discussed in Section 22.3.1), (6) use effective technical reviews as a filter prior to testing, (7) conduct technical reviews to assess the test strategy and test cases themselves, and (8) develop a continuous improvement approach (Chapter 37) for the testing process.

WebRef

An excellent list of testing resources can be found at **www.SQAtester.com.**

2 Gilb [Gil95] recommends that a software team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

22.3 Test Strategies for Conventional Software[3]

Many strategies can be used to test software. At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in the hope of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints all stakeholders. At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.

A testing strategy that is chosen by many software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units (sometimes on a daily basis), and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

22.3.1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

ADVICE

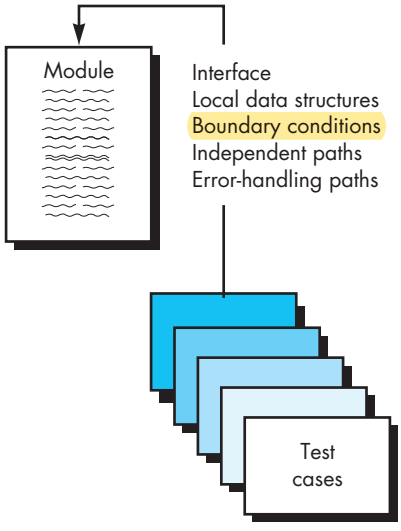
It's not a bad idea to design unit test cases before *you develop code for a component. It helps ensure that you'll develop code that will pass the tests.*

Unit Test Considerations. Unit tests are illustrated schematically in Figure 22.3. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

3 Throughout this book, we use the terms *conventional software* or *traditional software* to refer to common hierarchical or call-and-return software architectures that are frequently encountered in a variety of application domains. Traditional software architectures are not object oriented and do not encompass WebApps or MobileApps.

FIGURE 22.3
Unit test



? What errors are commonly found during unit testing?

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

ADVICE

Be sure that you design tests to execute every error-handling path. If you don't, the path may fail when it is invoked, exacerbating an already dicey situation.

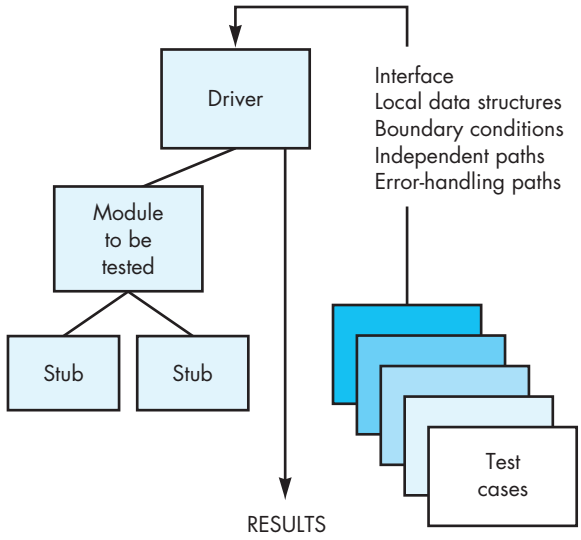
A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon [You75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test the error handling. If error-handling paths are implemented, they must be tested.

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-Test Procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the

FIGURE 22.4

Unit-test environment



categories discussed earlier. Each test case should be coupled with a set of expected results.

ADVICE

There are some situations in which you will not have the resources to do comprehensive unit testing. Select critical modules and those with high cyclomatic complexity and unit-test only those.

Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in Figure 22.4. In most applications a *driver* is nothing more than a "main program" that accepts test-case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent testing "overhead." That is, both are software that must be coded (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

22.3.2 Integration Testing

ADVICE

Taking the "big bang" approach to integration is a lazy strategy that is doomed to failure. Integrate incrementally, testing as you go.

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable

imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt nonincremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance and the entire program is tested as a whole. Chaos usually results! Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed.

ADVICE

When you develop a project schedule, you'll have to consider the manner in which integration will occur so that components will be available when needed.

Top-Down Integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

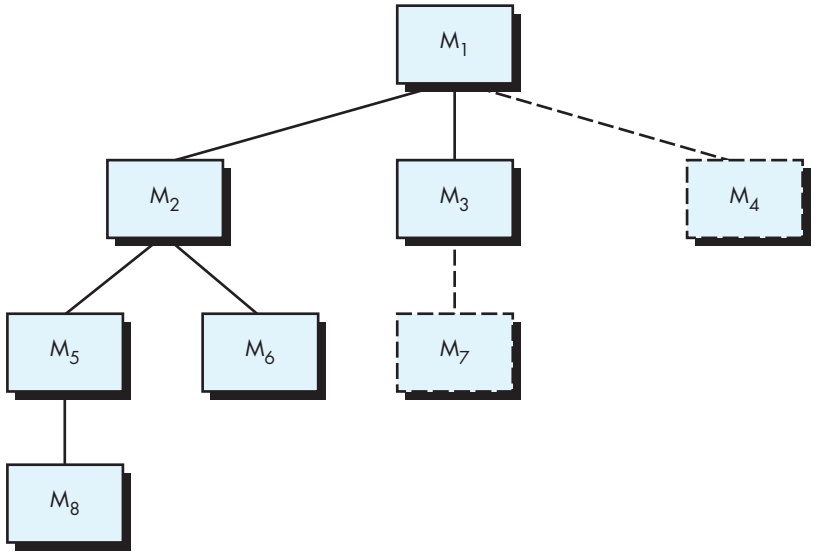
Referring to Figure 22.5, *depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows. The integration process is performed in a series of five steps:

? What are the steps for top-down integration?

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.

FIGURE 22.5

Top-down integration



5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a "well-factored" program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

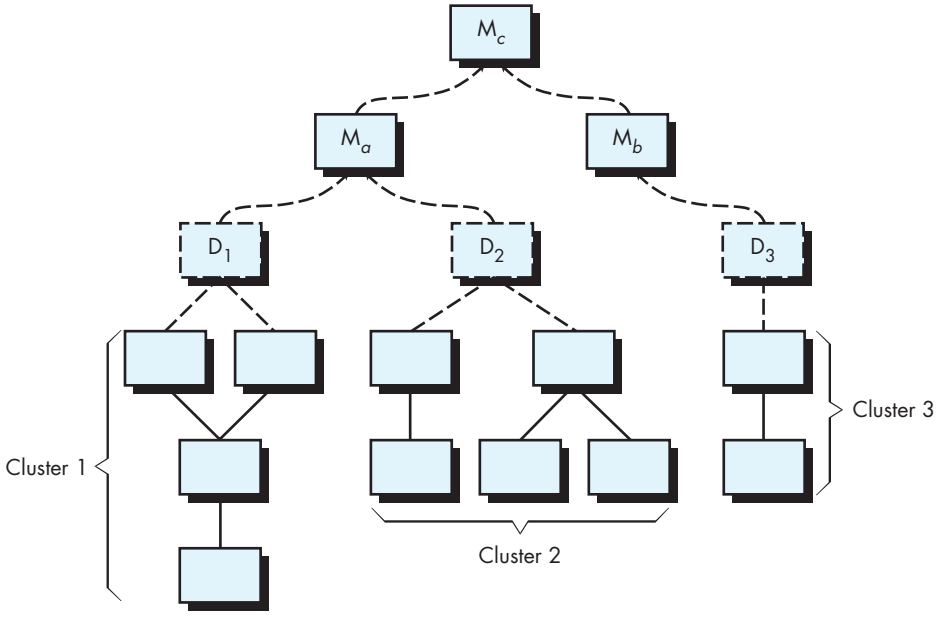
? What problems may be encountered when top-down integration is chosen?

Bottom-Up Integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

? What are the steps for bottom-up integration?

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

FIGURE 22.6
Bottom-up integration



KEY POINT
Bottom-up integration eliminates the need for complex stubs.

Integration follows the pattern illustrated in Figure 22.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D1 and D2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

ADVICE
Regression testing is an important strategy for reducing "side effects." Run regression tests every time a major change is made to the software (including the integration of new components).

Regression Testing. Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. Side effects associated with these changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent

playback and comparison. The *regression test suite* (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

KEY POINT

Smoke testing might be characterized as a rolling integration strategy. The software is rebuilt (with new components added) and smoke tested every day.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Smoke Testing. *Smoke testing* is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Quote:

"Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead."

Jim McCarthy

The daily frequency of testing gives both managers and practitioners a realistic assessment of integration testing progress. McConnell [McC96] describes the smoke test in the following manner:

> The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software projects:

? What benefits can be derived from smoke testing?

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.

- *The quality of the end product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

Integration Test Work Products. An overall plan for integration of the software and a description of specific tests is documented in a *Test Specification.* This work product incorporates a test plan and a test procedure and becomes part of the software configuration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for the *SafeHome* security system might be divided into the following test phases: user interaction, sensor processing, communications functions, and alarm processing.

Each of integration test phase delineates a broad functional category within the software and generally can be related to a specific domain within the software architecture. Therefore, program builds (groups of modules) are created to correspond to each phase.

A schedule for integration, the development of overhead software, and related topics are also discussed as part of the test plan. Start and end dates for each phase are established and "availability windows" for unit-tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results are also included.

A history of actual test results, problems, or peculiarities is recorded in a *Test Report* that can be appended to the *Test Specification,* if desired. Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

22.4 Test Strategies for Object-Oriented Software[4]

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics (Chapter 24).

22.4.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

KEY POINT

Class testing for OO software is analogous to module testing for conventional software. It is not advisable to test operations in isolation.

You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation *X* is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation *X*, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a stand-alone fashion (the conventional unit-testing approach) is usually ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

22.4.2 Integration Testing in the OO Context

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies (Section 22.3.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [Ber93].

4 Basic object-oriented concepts are presented in Appendix 2.

There are two different strategies for integration testing of OO systems [Bin94b]. The first, *thread-based testing*, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing*, begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes*, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

KEY POINT

An important strategy for integration testing of OO software is thread-based testing. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

22.5 Test Strategies for WebApps

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

KEY POINT

The overall strategy for WebApp testing can be summarized in the 10 steps noted here.

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.

WebRef

Excellent articles on WebApp testing can be found at **www.stickyminds.com/testing.asp**

9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for errors.

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by support staff who use regression tests derived from the tests developed when the WebApp was first engineered. Methods for WebApp testing are considered in Chapter 25.

22.6 Test Strategies for MobileApps

The strategy for testing mobile applications adopts the basic principles for all software testing. However, the unique nature of MobileApps demands the consideration of a number of specialized testing approaches:

- *User-experience testing*. Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices.
- *Device compatibility testing*. Testers verify that the MobileApp works correctly on all required hardware and software combinations.
- *Performance testing*. Testers check nonfunctional requirements unique to mobile devices (e.g., download times, processor speed, storage capacity, power availability).
- *Connectivity testing*. Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access.
- *Security testing*. Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users.
- *Testing-in-the-wild*. The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe.
- Certification testing. Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it.

Methods for MobileApp testing are considered in Chapter 26.

22.7 Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the

KEY POINT

Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user.

validation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a *Software Requirements Specification* has been developed, it describes all user-visible attributes of the software and contains a *Validation Criteria* section that forms the basis for a validation-testing approach.

22.7.1 Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability). If a deviation from specification is uncovered, a *deficiency list* is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

22.7.2 Configuration Review

An important element of the validation process is a *configuration review*. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit, is discussed in more detail in Chapter 29.

Quote:

"Given enough eyeballs, all bugs are shallow (e.g., given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone)."

E. Raymond

22.7.3 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software

product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

? What is the difference between an alpha test and a beta test?

The *alpha test* is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

A variation on beta testing, called *customer acceptance testing*, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SafeHome

Preparing for Validation

The scene: Doug Miller's office, as component-level design continues and construction of certain components continues.

The players: Doug Miller, software engineering manager, Vinod, Jamie, Ed, and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Doug: The first increment will be ready for validation in what . . . about three weeks?

Vinod: That's about right. Integration is going well. We're smoke testing daily, finding some bugs, but nothing we can't handle. So far, so good.

Doug: Talk to me about validation.

Shakira: Well, we'll use all of the use cases as the basis for our test design. I haven't started yet, but I'll be developing tests for all of the use cases that I've been responsible for.

Ed: Same here.

Jamie: Me too, but we've got to get our act together for acceptance test and also for alpha and beta testing, no?

Doug: Yes. In fact I've been thinking; we could bring in an outside contractor to help us with validation. I have the money in the budget . . . and it'd give us a new point of view.

Vinod: I think we've got it under control.

Doug: I'm sure you do, but an ITG gives us an independent look at the software.

Jamie: We're tight on time here, Doug. I for one don't have the time to baby-sit anybody you bring in to do the job.

Doug: I know, I know. But if an ITG works from requirements and use cases, not too much babysitting will be required.

Vinod: I still think we've got it under control.

Doug: I hear you, Vinod, but I am going to overrule on this one. Let's plan to meet with the ITG rep later this week. Get 'em started and see what they come up with.

Vinod: Okay, maybe it'll lighten the load a bit.

22.8 System Testing

Quote:

"Like death and taxes, testing is both unpleasant and inevitable."

Ed Yourdon

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

22.8.1 Recovery Testing

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

22.8.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [Bei84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more

than the value of the information that will be obtained. Security testing and security engineering are discussed in more detail in Chapter 27.

22.8.3 Stress Testing

Earlier software testing steps result in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

> **Quote:**
> "If you're trying to find true system bugs and you haven't subjected your software to a real stress test, then it's high time you started."
> **Boris Beizer**

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing*. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

22.8.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

22.8.5 Deployment Testing

In many cases, software must execute on a variety of platforms and under more than one operating system environment. *Deployment testing,* sometimes called *configuration testing,* exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and

specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

SOFTWARE TOOLS

Test Planning and Management

Objective: These tools assist a software team in planning the testing strategy that is chosen and managing the testing process as it is conducted.

Mechanics: Tools in this category address test planning, test storage, management and control, requirements traceability, integration, error tracking, and report generation. Project managers use them to supplement project scheduling tools. Testers use these tools to plan testing activities and to control the flow of information as the testing process proceeds.

Representative Tools:[5]

QaTraq Test Case Management Tool, developed by Traq Software (**www.testmanagement.com**), "encourages a structured approach to test management."

QAComplete, developed by SmartBear (http://smartbear.com/products/qa-tools/test-management), provides a single point of control for managing all phases of the agile testing process.

TestWorks, developed by Software Research (http://www.testworks.com/), contains a fully integrated suite of testing tools including tools for test management and reporting.

OpensourceTesting.org (**www.opensourcetesting.org/testmgt.php**) lists a variety of open-source test management and planning tools.

OpensourceTestManagement.com (**http://www.opensourcetestmanagement.com/**) lists a variety of open-source test management and planning tools.

22.9 THE ART OF DEBUGGING

Quote:

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

Maurice Wilkes, discovers debugging, 1949

Software testing is a process that can be systematically planned and specified. Test-case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. As a software engineer, you are often confronted with a "symptomatic" indication of a software problem as you evaluate the results of a test. That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

22.9.1 The Debugging Process

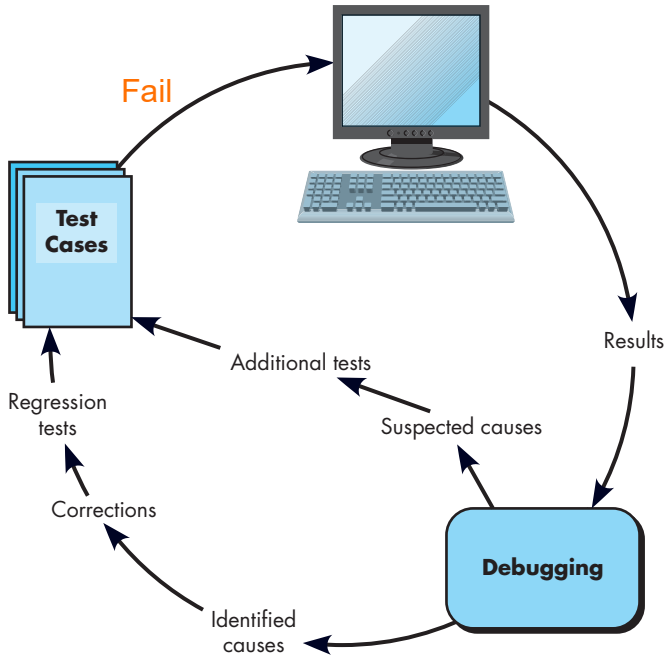
Debugging is not testing but often occurs as a consequence of testing.[6] Referring to Figure 22.7, the debugging process begins with the execution of a test case.

5 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

6 In making the statement, we take the broadest possible view of testing. Not only does the developer test software prior to release, but the customer/user tests the software every time it is used!

FIGURE 22.7

The debugging process



Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

? Why is debugging so difficult?

Why is debugging so difficult? In all likelihood, human psychology (see Section 22.9.2) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 12) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.

Quote:

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"

Brian Kernighan

5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces a software developer to fix one error and at the same time introduce two more.

ADVICE

Set a time limit, say two hours, on the amount of time you spend trying to debug a problem on your own. After that, get help!

22.9.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience. Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. We examine them in Section 22.9.3.

SAFEHOME

Debugging

The scene: Ed's cubical as code and unit testing is conducted.

The players: Ed and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Shakira (looking in through the entrance to the cubical): Hey . . . where were you at lunchtime?

Ed: Right here . . . working.

Shakira: You look miserable . . . what's the matter?

Ed (sighing audibly): I've been working on this bug since I discovered it at 9:30 this morning and it's what, 2:45 . . . I'm clueless.

Shakira: I thought we all agreed to spend no more than one hour debugging stuff on our own; then we get help, right?

Ed: Yeah, but . . .

Shakira (walking into the cubical): So what's the problem?

Ed: It's complicated, and besides, I've been looking at this for, what, 5 hours. You're not going to see it in 5 minutes.

Shakira: Indulge me . . . what's the problem?

[Ed explains the problem to Shakira, who looks at it for about 30 seconds without speaking, then . . .]

Shakira (a smile is gathering on her face): Uh, right there, the variable named *setAlarmCondition*. Shouldn't it be set to "false" before the loop gets started?

[Ed stares at the screen in disbelief, bends forward, and begins to bang his head gently against the monitor. Shakira, smiling broadly now, stands and walks out.]

22.9.3 Debugging Strategies

Regardless of the approach that is taken, debugging has one overriding objective—to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck.

In general, three debugging strategies have been proposed [Mye79]: brute force, backtracking, and cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Quote:

"The first step in fixing a broken program is getting it to fail repeatably (on the simplest example possible)."

T. Duff

Debugging Tactics. The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you'll find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated Debugging. Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted. Hailpern and Santhanam [Hai02] summarize the state of these tools when they note: ". . . many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TOOLS

Debugging

Objective: These tools provide automated assistance for those who must debug software problems. The intent is to provide insight that may be difficult to obtain if approaching the debugging process manually.

Mechanics: Most debugging tools are programming language and environment specific.

Representative Tools:[7]

Borland Silkt, distributed by Borland (**http://www.borland.com/products/**), assists in both testing and debugging.

Coverty Development Testing Platform, developed by Coverty (**http://www.coverity.com/products/**), provides a means of introducing quality and security testing into the early development process.

C++Test, developed by Parasoft (www.parasoft.com), is a unit-testing tool that supports a full range of tests on C and C++ code. Debugging features assist in the diagnosis of errors that are found.

CodeMedic, developed by NewPlanet Software (www.newplanetsoftware.com/medic/), provides a graphical interface for the standard UNIX debugger, *gdb,* and implements its most important features. *gdb* currently supports C/C++, Java, PalmOS, various embedded systems, assembly language, FORTRAN, and Modula-2.

GNATS, a freeware application (www.gnu.org/software/gnats/), is a set of tools for tracking bug reports.

The People Factor. Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.[8] A final maxim for debugging might be: "When all else fails, get help!"

22.9.4 Correcting the Error

Quote:

"The best tester isn't the one who finds the most bugs . . . the best tester is the one who gets the most bugs fixed."

Cem Kaner et al.

Once a bug has been found, it must be corrected. But as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [Van89] suggests three simple questions that you should ask before making the "correction" that removes the cause of a bug:

1. *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.
2. *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8 The concept of pair programming (recommended as part of the Extreme Programming model discussed in Chapter 5) provides a mechanism for "debugging" as the software is designed and coded.

is to be made in a highly coupled section of the program, special care must be taken when any change is made.

3. *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach (Chapter 21). If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

22.10 Summary

Software testing accounts for the largest percentage of technical effort in the software process. Regardless of the type of software you build, a strategy for systematic test planning, execution, and control begins by considering small elements of the software and moves outward toward the program as a whole.

The objective of software testing is to uncover errors. For conventional software, this objective is achieved through a series of test steps. Unit and integration tests concentrate on functional verification of a component and incorporation of components into the software architecture. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

The strategy for testing object-oriented software begins with tests that exercise the operations within a class and then moves to thread-based testing for integration. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

Web and MobileApps are tested in much the same way as OO systems. However, tests are designed to exercise content, functionality, the interface, navigation, and other aspects of app performance and security. MobileApps require specialized testing approaches that focus on testing the App on multiple devices and in real world network environments.

Unlike testing (a systematic, planned activity), debugging can be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

Problems and Points to Ponder

22.1 Using your own words, describe the difference between verification and validation. Do both make use of test-case design methods and testing strategies?

22.2 List some problems that might be associated with the creation of an independent test group. Are an ITG and an SQA group made up of the same people?

22.3 Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 22.1.3? What possible complications might arise for embedded systems?

22.4 Why is a highly coupled module difficult to unit test?

22.5 The concept of "antibugging" (Section 22.2.1) is an extremely effective way to provide built-in debugging assistance when an error is uncovered:

a. Develop a set of guidelines for antibugging.
b. Discuss advantages of using the technique.
c. Discuss disadvantages.

22.6 How can project scheduling affect integration testing?

22.7 Is unit testing possible or even desirable in all circumstances? Provide examples to justify your answer.

22.8 Who should perform the validation test—the software developer or the software user? Justify your answer.

22.9 Develop a complete test strategy for the *SafeHome* system discussed earlier in this book. Document it in a *Test Specification*.

22.10 As a class project, develop a *Debugging Guide* for your installation. The guide should provide language and system-oriented hints that you have learned through the school of hard knocks! Begin with an outline of topics that will be reviewed by the class and your instructor. Publish the guide for others in your local environment.

Further Readings and Information Sources

Virtually every book on software testing discusses strategies along with methods for test-case design. Whittaker (*How Google Tests Software,* Addison-Wesley, 2012) and (*How to Break Software,* Addison-Wesley, 2002), Spiller and his colleagues (*Software Testing Foundations,* Rocky Nook, 2011), Black (*Managing the Testing,* 3rd ed., Wiley, 2009) and (*Pragmatic Software Testing,* Wiley, 2007), Page and his colleagues (*How We Test Software at Microsoft,* Microsoft Press, 2008), Lewis (*Software Testing and Continuous Quality Improvement,* 3rd ed., Auerbach, 2008), Everett and Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Perry (*Effective Methods for Software Testing,* 3rd ed., Wiley, 2005), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (Systematic Software Testing, Artech House, 2002), and Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

For those readers with an interest in agile software development methods, Gartner and Gartner (*ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*, Addison-Wesley, 2012), Crispin and Gregory (*Agile Testing: A Practical Guide for Testers and Teams*, Addison-Wesley, 2009), Crispin and House (*Testing Extreme Programming,* Addison-Wesley, 2002) and Beck (*Test Driven Development: By Example,* Addison-Wesley, 2002) present testing strategies and tactics for Extreme Programming. Kamer and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) present a collection of more than 300 pragmatic "lessons" (guidelines) that every software tester should learn. Watkins (*Testing IT: An Off-the-Shelf Testing Process*, 2nd ed. Cambridge University Press, 2010) establishes an effective testing framework for all types of developed and acquired software. Manges and

O'Brien (*Agile Testing with Ruby and Rails,* Apress, 2008) addresses testing strategies and techniques for the Ruby programming language and Web framework.

Bashir and Goel (*Testing Object-Oriented Software,* Springer-Verlag, 2012), Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), Kung and his colleagues (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998), and Marick (*The Craft of Software Testing,* Prentice Hall, 1997) present strategies and methods for testing OO systems.

Guidelines for debugging are contained in books by Grötker and his colleagues (*The Developer's Guide to Debugging,* 2nd ed., CreateSpace Independent Publishing, 2012), Whittaker (*Exploratory Testing,* Addison-Wesley, 2009), Zeller (*Why Programs Fail: A Guide to Systematic Debugging,* 2nd ed., Morgan Kaufmann, 2009), Butcher (*Debug I!,* Pragmatic Bookshelf, 2009), Agans (*Debugging,* Amacon, 2006), and Tells and Hsieh (*The Science of Debugging,* The Coreolis Group, 2001). Kaspersky (*Hacker Debugging Uncovered,* A-List Publishing, 2005) addresses the technology of debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice Hall, 2002) presents techniques for managing defects that are encountered in object-oriented systems. Beizer [Bei84] presents an interesting "taxonomy of bugs" that can lead to effective methods for test planning.

Books by Graham and Fewster (*Experiences of Test Automation,* Addison-Wesley, 2012) and Dustin and his colleagues (*Implementing Automated Software Testing*, Addison-Wesley, 2009) discuss automated testing. Books by Hunt and John (*Java Performance,* Addison-Wesley, 2011), Hewardt and his colleagues (*Advanced .NET Debugging,* Addison-Wesley, 2009), Matloff and his colleagues (*The Art of Debugging with GDB, DDD, and Eclipse,* No Starch Press, 2008), Madisetti and Akgul (*Debugging Embedded Systems,* Springer, 2007), Robbins (*Debugging Microsoft .NET 2.0 Applications,* Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning,* Prentice Hall, 2005), Ford and Teorey (*Practical Debugging in C++,* Prentice Hall, 2002), Brown (*Debugging Perl,* McGraw-Hill, 2000), and Mitchell (*Debugging Java,* McGraw-Hill, 2000) address the special nature of debugging for the environments implied by their titles.

A wide variety of information sources on software testing strategies are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software testing strategies can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

23 Testing Conventional Applications

Key Concepts

basis path testing . 500
black-box testing . 509
boundary value analysis 512
control structure testing 507
cyclomatic complexity 503
equivalence partitioning 511
flow graph 500
graph matrices . . . 506

Testing presents an interesting dilemma for software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Beizer [Bei90] describes this situation effectively when he states:

> There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top-down design, . . . then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission

Quick Look

What is it? Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic and interfaces of every software component and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Why is it important? Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. To find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? For conventional applications, software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test-case design techniques and (2) software requirements are exercised using "black box" test-case design techniques. Use cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.

What is the work product? A set of test cases designed to exercise internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness. In addition, you can evaluate test coverage and track error detection activities.

graph-based testing methods 509
model-based testing 516
orthogonal array testing 513
patterns 519
white-box testing . 500

> of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic? For not solving human communications problems that have been kicked around . . . for forty centuries?

Should testing instill guilt? Is testing really destructive? The answer to these questions is "No!"

In this chapter, we discuss techniques for software test-case design for conventional applications. Test-case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives and the testing strategies discussed in Chapter 22.

23.1 Software Testing Fundamentals

Quote:

"Every program does something right, it just may not be the thing we want it to do."

Author unknown

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach[1] provides the following definition for testability: "*Software testability* is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

Operability. "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

? What are the characteristics of testability?

Observability. "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and

1 The paragraphs that follow are used with permission of James Bach (copyright 1994) and have been adapted from material that originally appeared in a posting in the newsgroup comp.software-eng.

hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements); *structural simplicity* (e.g., architecture is modularized to limit the propagation of faults), and *code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.

Quote:

"Errors are more common, more pervasive, and more troublesome in software than with other technologies."

David Parnas

Understandability. "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

You can use the attributes suggested by Bach to develop software work products that are amenable to testing.

Test Characteristics. And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

? What is a "good" test?

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

SafeHome

Designing Unique Tests

The scene: Vinod's cubical.

The players: Vinod and Ed—members of the *SafeHome* software engineering team.

The conversation:

Vinod: So these are the test cases you intend to run for the *passwordValidation* operation.

Ed: Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.

Vinod: So let's see . . . you note that the correct password will be 8080, right?

Ed: Uh huh.

Vinod: And you specify passwords 1234 and 6789 to test for error in recognizing invalid passwords?

Ed: Right, and I also test passwords that are close to the correct password, see . . . 8081 and 8180.

Vinod: Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

Ed: Well, they're different values.

Vinod: That's true, but if 1234 doesn't uncover an error . . . in other words . . . the *passwordValidation* operation notes that it's an invalid password, it's not likely that 6789 will show us anything new.

Ed: I see what you mean.

Vinod: I'm not trying to be picky here . . . it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.

Ed: Not a problem . . . I'll give this a bit more thought.

23.2 Internal and External Views of Testing

Quote:

"There is only one rule in designing test cases: cover all features, but do not make too many test cases."

Tsuneo Yamaura

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.[2]

KEY POINT

White-box tests can be designed only after component-level design (or source code) exists. The logical details of the program must be available.

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software. *White-box testing* of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths,

2 The terms *functional testing* and *structural testing* are sometimes used in place of black-box and white-box testing, respectively.

develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

INFO

Exhaustive Testing

Consider a 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules.

Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

23.3 White-Box Testing

Quote:

"Bugs lurk in corners and congregate at boundaries."

Boris Beizer

White-box testing, sometimes called *glass-box testing* or *structural testing*, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

23.4 Basis Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe [McC76]. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

23.4.1 Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be

FIGURE 23.1

Flow graph notation

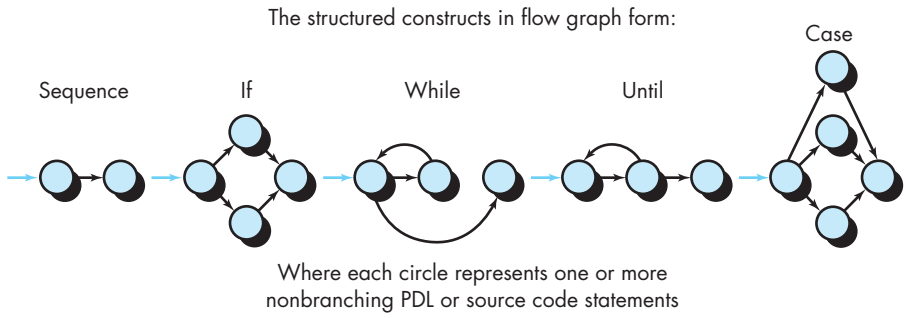
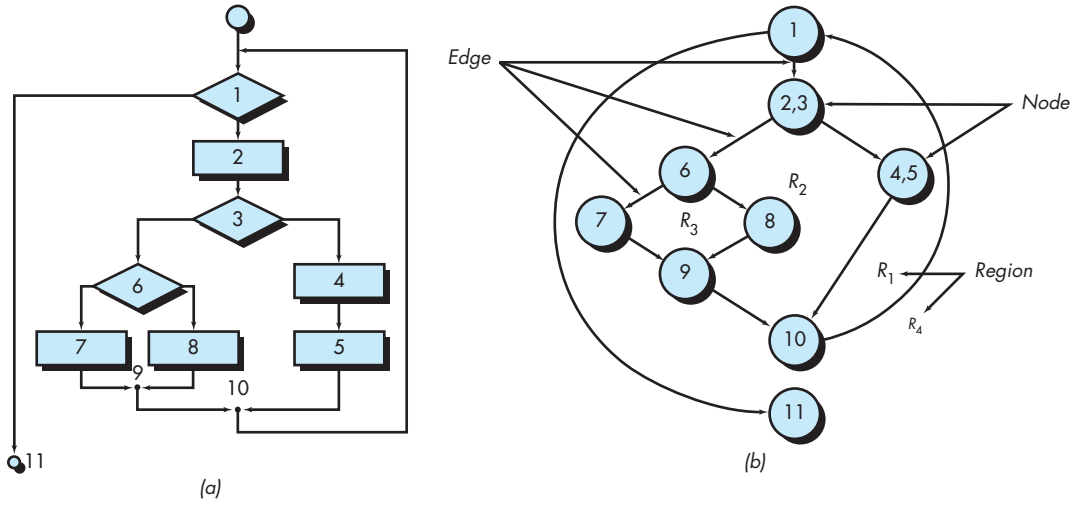


FIGURE 23.2 **(a) Flowchart and (b) flow graph**



introduced.[3] The flow graph depicts logical control flow using the notation illustrated in Figure 23.1. Each structured construct (Chapter 14) has a corresponding flow graph symbol.

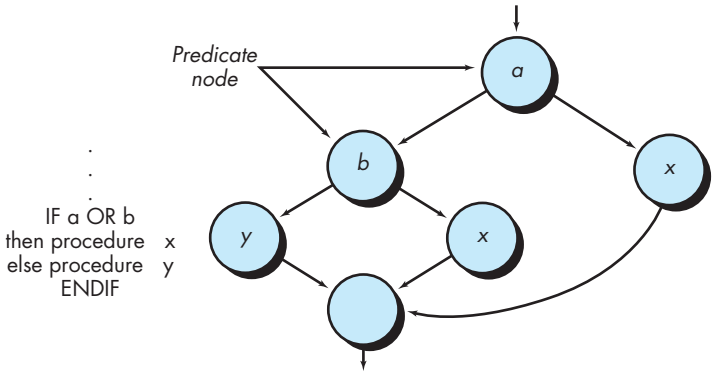
ADVICE

A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allows you to trace program paths more readily.

To illustrate the use of a flow graph, consider the procedural design representation in Figure 23.2a. Here, a flowchart is used to depict program control structure. Figure 23.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 23.2b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent

3 In actuality, the basis path method can be conducted without the use of flow graphs. However, they serve as a useful notation for understanding control flow and illustrating the approach.

FIGURE 23.3
Compound logic



any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.[4]

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 23.3, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

23.4.2 Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 23.2b is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

4 A more detailed discussion of graphs and their uses is presented in Section 23.6.1.

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 23.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

ADVICE

Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. Use it for test planning as well as test-case design.

How do you know how many paths to look for? The computation of cyclomatic complexity provides the answer. *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

? How do I compute cyclomatic complexity?

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

   $$V(G) = E - N + 2$$

   where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $$V(G) = P + 1$$

   where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 23.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

KEY POINT

Cyclomatic complexity provides the upper bound on the number of test cases that will be required to guarantee that every statement in the program has been executed at least one time.

1. The flow graph has four regions.
2. $V(G) = 11$ edges $- 9$ nodes $+ 2 = 4$.
3. $V(G) = 3$ predicate nodes $+ 1 = 4$.

Therefore, the cyclomatic complexity of the flow graph in Figure 23.2b is 4.

More important, the value for $V(G)$ provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

SafeHome

Using Cyclomatic Complexity

The scene: Shakira's cubicle.

The players: Vinod and Shakira—members of the *SafeHome* software engineering team who are working on test planning for the security function.

The conversation:

Shakira: Look . . . I know that we should unit-test all the components for the security function, but there are a lot of 'em and if you consider the number of operations that have to be exercised, I don't know . . . maybe we should forget white-box testing, integrate everything, and start running black-box tests.

Vinod: You figure we don't have enough time to do component tests, exercise the operations, and then integrate?

Shakira: The deadline for the first increment is getting closer than I'd like . . . yeah, I'm concerned.

Vinod: Why don't you at least run white-box tests on the operations that are likely to be the most error prone?

Shakira (exasperated): And exactly how do I know which are the most error prone?

Vinod: *V* of *G*.

Shakira: Huh?

Vinod: Cyclomatic complexity—*V* of *G*. Just compute *V*(*G*) for each of the operations within each of the components and see which have the highest values for *V*(*G*). They're the ones that are most likely to be error prone.

Shakira: And how do I compute *V* of *G*?

Vinod: It's really easy. Here's a book that describes how to do it.

Shakira (leafing through the pages): Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V*(*G*) will be the candidates for white-box tests.

Vinod: Just remember that there are no guarantees. A component with a low *V*(*G*) can still be error prone.

Shakira: Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

23.4.3 Deriving Test Cases

Quote:

"To err is human, to find a bug is divine."

Robert Dunn

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average*, depicted in PDL in Figure 23.4, will be used as an example to illustrate each step in the test-case design method. Note that *average*, although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. **Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules presented in Section 23.4.1. Referring to the PDL for *average* in Figure 23.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 23.5.
2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity *V*(*G*) is determined by applying the algorithms described in Section 23.4.2. It should be noted that *V*(*G*) can be determined

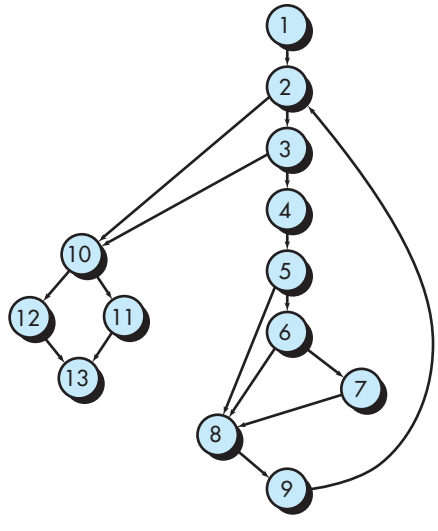
FIGURE 23.4

PDL with nodes identified

```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes the
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
1   total.input = total.valid = 0;                                  2
    sum = 0;
    DO WHILE value[i] <> –999 AND total.input < 100                 3
4   increment total.input by 1;
    IF value[i] > = minimum AND value[i] < = maximum                6
5       THEN increment total.valid by 1;
7               sum = s sum + value[i]
        ELSE skip
8   ENDIF
    increment i by 1;
9   ENDDO
    IF total.valid > 0                                              10
11  THEN average = sum / total.valid;
12      ELSE average = –999;
13  ENDIF
END average
```

FIGURE 23.5

Flow graph for the procedure average



without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and adding 1. Referring to Figure 23.5,

$V(G) = 6$ regions

$V(G) = 17 \text{ edges} - 13 \text{ nodes} + 2 = 6$

$V(G) = 5 \text{ predicate nodes} + 1 = 6$

Quote:

"The Ariane 5 rocket blew up on lift-off due solely to a software defect (a bug) involving the conversion of a 64-bit floating point value into a 16-bit integer. The rocket and its four satellites were *uninsured* and worth $500 million. [Path tests that exercised the conversion path] would have found the bug but were vetoed for budgetary reasons."

A news report

3. **Determine a basis set of linearly independent paths.** The value of $V(G)$ provides the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

   Path 1: 1-2-10-11-13
   Path 2: 1-2-10-12-13
   Path 3: 1-2-3-10-11-13
   Path 4: 1-2-3-4-5-8-9-2-. . .
   Path 5: 1-2-3-4-5-6-8-9-2-. . .
   Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

   The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. **Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

23.4.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. A data structure, called a *graph matrix,* can be quite useful for developing a software tool that assists in basis path testing.

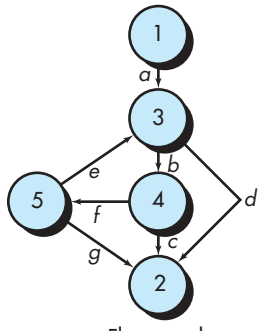
A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [Bei90] is shown in Figure 23.6.

? What is a graph matrix and how do we extend it for use in testing?

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*.

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph

FIGURE 23.6
Graph matrix



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

Beizer [Bei90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

23.5 CONTROL STRUCTURE TESTING

Quote:

"Paying more attention to running tests than to designing them is a classic mistake."

Brian Marick

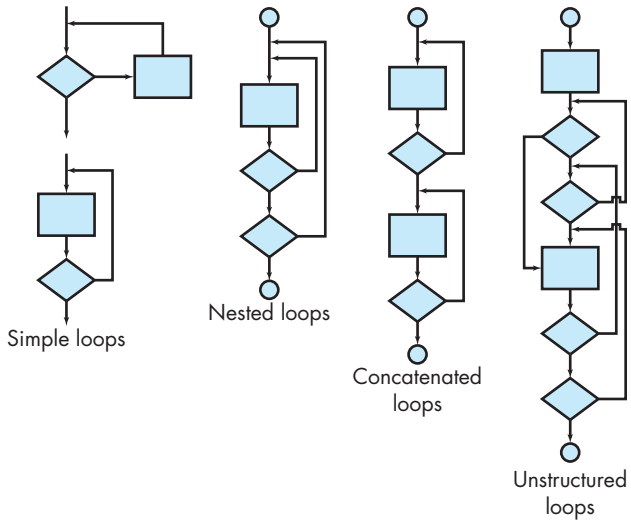
The basis path testing technique described in Section 23.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. *Data flow testing* [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 23.7).

FIGURE 23.7

Classes of Loops



Quote:

"Good testers are masters at noticing 'something funny' and acting on it."

Brian Marick

Simple Loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n - 1$, n, $n + 1$ passes through the loop.

Nested Loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

ADVICE

You can't test unstructured loops effectively. Refactor them.

Concatenated Loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured Loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 14).

23.6 Black-Box Testing

Black-box testing, also called *behavioral testing* or *functional testing*, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing (see Chapter 22). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

? What questions do black-box tests answer?

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria [Mye79]: test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

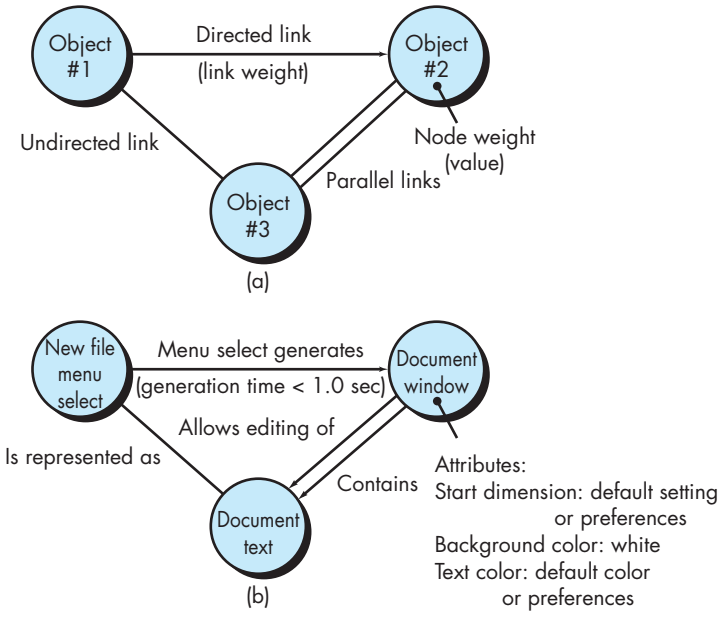
23.6.1 Graph-Based Testing Methods

The first step in black-box testing is to understand the objects[5] that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects

5 In this context, you should consider the term *objects* in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

FIGURE 23.8

(a) Graph notation; (b) simple example



have the expected relationship to one another" [Bei95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

KEY POINT

A graph represents the relationships between data objects and program objects, enabling us to derive test cases that search for errors associated with these relationships.

To accomplish these steps, you begin by creating a *graph*—a collection of *nodes* that represent objects, *links* that represent the relationships between objects, *node weights* that describe the properties of a node (e.g., a specific data value or state behavior), and *link weights* that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure 23.8a. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 23.8b) where

Object #1 = **newFile** (menu selection)

Object #2 = **documentWindow**

Object #3 = **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes

that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [Bei95] describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps. For example, a data object **flightInformationInput** is followed by the operation *validationAvailabilityProcessing*().

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **orderInformation** is verified during *inventoryAvailabilityLook-up()* and is followed by **customerBilling-Information** input). The state diagram (Chapter 11) can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node **FICATaxWithheld** (**FTW**) is computed from gross wages (**GW**) using the relationship, **FTW = 0.62 × GW.**

Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. If you have further interest, see [Bei95] for a comprehensive coverage.

23.6.2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the

preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

? How do I define equivalence classes for testing?

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Quote:

"An effective way to test code is to exercise it at its natural boundaries."

Brian Kernighan

23.6.3 Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

KEY POINT

BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class.

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an

output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

23.6.4 Orthogonal Array Testing

KEY POINT

Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.

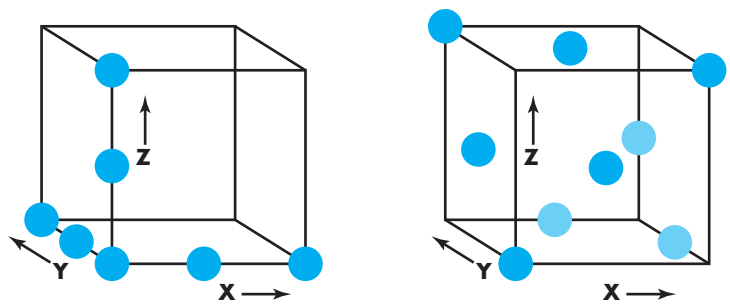
There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X, Y,* and *Z*. Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke [Pha97] suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 23.9. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

FIGURE 23.9

A geometric view of test cases
Source: [Pha97].



One input item at a time **L9 orthogonal array**

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property" [Pha97]. That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 23.9. Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). But these would uncover only *single mode faults* [Pha97], that is, faults that are triggered by a single parameter.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is $3^4 = 81$, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax *send* function is illustrated in Figure 23.10.

FIGURE 23.10

An L9 orthogonal array

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke [Pha97] assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 23.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault*. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

You can find a detailed discussion of orthogonal array testing in [Pha89].

SOFTWARE TOOLS

Test-Case Design

Objective: To assist the software team in developing a complete set of test cases for both black-box and white-box testing.

Mechanics: These tools fall into two broad categories: static testing tools and dynamic testing tools. Three different types of static testing tools are used in the industry: code-based testing tools, specialized testing languages, and requirements-based testing tools. Code-based testing tools accept source code as input and perform a number of analyses that result in the generation of test cases. Specialized testing languages (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. Requirements-based testing tools isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements. Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program.

Representative Tools:[6]

McCabeTest, developed by McCabe & Associates (**www.mccabe.com**), implements a variety of path testing techniques derived from an assessment of cyclomatic complexity and other software metrics.

TestWorks, developed by Software Research (**http://www.testworks.com/stwhome.html**), is a complete set of automated testing tools that assists in the design of tests cases for software developed in C/C++ and Java and provides support for regression testing.

T-VEC Test Generation System, developed by T-VEC Technologies (**www.t-vec.com**), is a tool set that supports unit, integration, and validation testing by assisting in the design of test cases using information contained in an OO requirements specification.

e-TEST Suite, developed by Empirix (**www.empirix.com**), encompasses a complete set of tools for testing WebApps, including tools that assist test-case design and test planning.

6 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

23.7 Model-Based Testing

Quote:

"It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."

Steve McConnell

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases [DAC03]. In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model (Chapter 11), as the basis for the design of test cases.[7] The MBT technique requires five steps:

1. **Analyze an existing behavioral model for the software or create one.** Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the steps discussed in Chapter 11: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system (e.g., see Figure 11.1), and (5) review the behavioral model to verify accuracy and consistency.
2. **Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.** The inputs will trigger events that will cause the transition to occur.
3. **Review the behavioral model and note the expected outputs as the software makes the transition from state to state.** Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model.
4. **Execute the test cases.** Tests can be executed manually or a test script can be created and executed using a testing tool.
5. **Compare actual and expected results and take corrective action as required.**

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

23.8 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. But errors in help facilities or program documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more

7 Model-based testing can also be used when software requirements are represented with decision tables, grammars, or Markov chains [DAC03].

frustrating than following a user guide or an online help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that documentation testing should be a meaningful part of every software test plan.

Documentation testing can be approached in two phases. The first phase, technical review (Chapter 20), examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

INFO

Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document's table of contents and index robust, accurate, and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

23.9 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test-case designer have to consider conventional test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided

when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the *copying* state). These same operator interrupts, if input when the machine is in the *jammed* state, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

An overall four-step strategy for real-time software testing can be proposed:

? What is an effective strategy for testing a real-time system?

- **Task testing.** Test each task independently. That is, conventional tests are designed for each task and executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.
- **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to equivalence partitioning (Section 23.6.2), events (e.g., interrupts, control signals) are categorized for testing. For example, events for the photocopier might be user interrupts (e.g., reset counter), mechanical interrupts (e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated). Each of these events is tested individually, and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events.
- **Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
- **System testing.** Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software-hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state diagram (Chapter 11), the tester develops a list of all possible interrupts and the processing that occurs as a consequence of

the interrupts. Tests are then designed to assess the following system characteristics:

- Are interrupt priorities properly assigned and properly handled?
- Is processing for each interrupt handled correctly?
- Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
- Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

23.10 Patterns for Software Testing

WebRef

A software testing patterns catalog can be found at **http://c2.com/cgi-bin/wiki?TestingPatterns.**

The use of patterns as a mechanism for describing solutions to specific design problems was discussed in Chapter 16. But patterns can also be used to propose solutions to other software engineering situations—in this case, software testing. *Testing patterns* describe common testing problems and solutions that can assist you in dealing with them.

Much of software testing, even during the past decade, has been an ad hoc activity. If testing patterns can help a software team to communicate about testing more effectively, to understand the motivating forces that lead to a specific approach to testing, and to approach the design of tests as an evolutionary activity in which each iteration results in a more complete suite of test cases, then patterns have accomplished much.

KEY POINT

Testing patterns can help a software team communicate more effectively about testing and better understand the forces that lead to a specific testing approach.

Testing patterns are described in much the same way as design patterns (Chapter 16). Dozens of testing patterns have been proposed in the literature (e.g., [Mar02]). The following three testing patterns (presented in abstract form only) provide representative examples:

Pattern name: **PairTesting**

Abstract: A process-oriented pattern, **pair testing** describes a technique that is analogous to pair programming (Chapter 5) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Pattern name: **SeparateTestInterface**

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The **SeparateTestInterface** pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component" [Lan01].

Pattern name: **ScenarioTesting**

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement [Kan01].

A comprehensive discussion of testing patterns is beyond the scope of this book. If you have further interest, see [Bin99], [Mar02], and [Tho04] for additional information on this important topic.

23.11 Summary

The primary objective for test-case design is to derive a set of tests that have the highest likelihood for uncovering errors in software. To accomplish this objective, two different categories of test-case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Hetzel [Het84] describes white-box testing as "testing in the small." His implication is that the white-box tests that have been considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens your focus and might be called "testing in the large."

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters. Model-based testing uses elements of the requirements model to test the behavior of an application.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your

customer uses the program, a test is being conducted." By applying test-case design, you can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

Problems and Points to Ponder

23.1. Myers [Mye79] uses the following program as a self-assessment for your ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

23.2. Design and implement the program (with error handling where appropriate) specified in Problem 23.1. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.

23.3. Can you think of any additional testing objectives that are not discussed in Section 23.1.1?

23.4. Select a software component that you have designed and implemented recently. Design a set of test cases that will ensure that all statements have been executed using basis path testing.

23.5. Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language of your choice. Use the graph matrix as the operative data structure in your design.

23.6. Read Beizer [Bei95] or a related Web-based source (e.g., **www.laynetworks.com/Discrete%20Mathematics_1g.htm**) and determine how the program you have developed in Problem 23.5 can be extended to accommodate various link weights. Extend your tool to process execution probabilities or link processing times.

23.7. Design an automated tool that will recognize loops and categorize them as indicated in Section 23.5.3.

23.8. Extend the tool described in Problem 23.7 to generate test cases for each loop category, once encountered. It will be necessary to perform this function interactively with the tester.

23.9. Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.

23.10. Will exhaustive testing (even if it is possible for very small programs) guarantee that the program is 100 percent correct?

23.11. Test a user manual (or help facility) for an application that you use frequently. Find at least one error in the documentation.

Further Readings and Information Sources

Virtually all books dedicated to software testing consider both strategy and tactics. Therefore, further readings noted for Chapter 22 are equally applicable for this chapter. Burnstein (*Practical Software Testing,* Springer, 2010), Crispin and Gregory (*Agile Testing: A Practical Guide for Testers and Agile Teams,* Addison-Wesley, 2009), Lewis (*Software Testing*

and Continuous Quality Improvement, 3rd ed., Auerbach, 2008), Ammann and Offutt (*Introduction to Software Testing,* Cambridge University Press, 2008), Everett and McCleod (Software Testing, Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller and his colleagues (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3rd ed., Wiley, 2006), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), and Whittaker (*Exploratory Software Testing: Tips, Tricks, and Techniques to Guide Test Design,* Addison-Wesley, 2009) and (*How to Break Software,* Addison-Wesley, 2002) are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

A third edition of Myers's [Mye79] classic text has been produced by Myers and his colleagues (*The Art of Software Testing,* 3rd ed., Wiley, 2011) and covers test-case design techniques in considerable detail. Black (*Managing the Testing Process*, 3rd ed., Wiley, 2009), Jorgensen (*Software Testing: A Craftsman's Approach,* 3rd ed., CRC Press, 2008), Pezze and Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3rd ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), and Hutcheson (*Software Testing Fundamentals,* Wiley, 2003) each provide useful presentations of test-case design methods and techniques. Beizer's [Bei90] classic text provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [Bei95] presents a concise treatment of important methods.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Graham and her colleagues (*Experiences of Test Automation: Case Studies of Software Test Automation,* Addison-Wesley, 2012) and (*Software Test Automation*, Addison-Wesley, 1999), Li and Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely and Posey (*Just Enough Software Test Automation,* Prentice Hall, 2002); Dustin, Rashka, and Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); and Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing. Nquyen and his colleagues (*Happy About Global Software Test Automation,* Happy About Press, 2006) present an executive overview of testing automation.

Meszaros (*Unit Test Patterns: Refactoring Test Code,* Addison-Wesley, 2007), Thomas and his colleagues (*Java Testing Patterns,* Wiley, 2004) and Binder [Bin99] describe testing patterns that cover testing of methods, classes/clusters, subsystems, reusable components, frameworks, and systems as well as test automation and specialized database testing.

A wide variety of information sources on test-case design methods are available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing techniques can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

24

Testing Object-Oriented Applications

Key Concepts

class testing 528
cluster testing. . . . 529
consistency. 526
fault-based testing 531
multiple class testing 534
object-oriented models 525
partition testing . . 533
random testing . . . 532
scenario-based testing 532

In Chapter 23, we noted that the objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of OO programs changes both testing strategy and testing tactics.

It could be argued that as reusable class libraries grow in size, greater reuse will mitigate the need for heavy testing of OO systems. Exactly the opposite is true. Binder [Bin94b] discusses this when he states:

> [E]ach reuse is a new context of usage and retesting is prudent. It seems likely that more, not less testing will be needed to obtain high reliability in object-oriented systems.

Quick Look

What is it? The architecture of object-oriented (OO) software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) performs functions that help to achieve system requirements. It is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Who does it? Object-oriented testing is performed by software engineers and testing specialists.

Why is it important? You have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? OO testing is strategically analogous to the testing of conventional systems, but it is tactically different. Because the OO analysis and design models are similar in structure and content to the resultant OO program, "testing" is initiated with the review of these models. Once code has been generated, OO testing begins "in the small" with class testing. A series of tests are designed that exercise class operations and examine whether errors exist as one class collaborates with other classes. As classes are integrated to form a subsystem, thread-based, use-based, and cluster testing along with fault-based approaches are applied to fully exercise collaborating classes. Finally, use cases (developed as part of the analysis model) are used to uncover errors at the software validation level.

What is the work product? A set of test cases, designed to exercise classes, their collaborations, and behaviors is designed and documented; expected results are defined, and actual results are recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion, and review the tests cases you do create for thoroughness.

testing methods . . 529
testing strategies . 528
thread-based testing 529
use-based testing . 529

To adequately test OO systems, three things must be done: (1) the definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models, (2) the strategy for unit and integration testing must change significantly, and (3) the design of test cases must account for the unique characteristics of OO software.

24.1 Broadening the View of Testing

The construction of object-oriented software begins with the creation of analysis and design models.[1] Because of the evolutionary nature of the OO software engineering paradigm, these models begin as relatively informal representations of system requirements and evolve into detailed models of classes, class relationships, system design and allocation, and object design (incorporating a model of object connectivity via messaging). At each stage, the models can be "tested" in an attempt to uncover errors prior to their propagation to the next iteration.

ADVICE

Although the review of the OO analysis and design models is an integral part of "testing" an OO application, recognize that it is not sufficient in and of itself. You must conduct executable tests as well.

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code levels. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

For example, consider a class in which a number of attributes are defined during the first iteration of analysis. An extraneous attribute is appended to the class (due to a misunderstanding of the problem domain). Two operations are then specified to manipulate the attribute. A review is conducted and a domain expert points out the problem. By eliminating the extraneous attribute at this stage, the following problems and unnecessary effort may be avoided during analysis:

1. Special subclasses may have been generated to accommodate the unnecessary attribute or exceptions to it. Work involved in the creation of unnecessary subclasses has been avoided.
2. A misinterpretation of the class definition may lead to incorrect or extraneous class relationships.
3. The behavior of the system or its classes may be improperly characterized to accommodate the extraneous attribute.

1 Analysis and design modeling techniques are presented in Part 2 of this book. Basic OO concepts are presented in Appendix 2.

If the problem is not uncovered during analysis and propagated further, the following problems could occur (and will have been avoided because of the earlier review) during design:

1. Improper allocation of the class to subsystem and/or tasks may occur during system design.
2. Unnecessary design work may be expended to create the procedural design for the operations that address the extraneous attribute.
3. The messaging model will be incorrect (because messages must be designed for the operations that are extraneous).

Quote:

"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities."

Edsger Dijkstra

If the problem remains undetected during design and passes into coding, considerable effort will be expended to generate code that implements an unnecessary attribute, two unnecessary operations, messages that drive interobject communication, and many other related issues. In addition, testing of the class will absorb more time than necessary. Once the problem is finally uncovered, modification of the system must be carried out with the ever-present potential for side effects that are caused by change.

During latter stages of their development, object-oriented analysis (OOA) and design (OOD) models provide substantial information about the structure and behavior of the system. For this reason, these models should be subjected to rigorous review prior to the generation of code.

All object-oriented models should be tested (in this context, the term *testing* incorporates technical reviews) for correctness, completeness, and consistency within the context of the model's syntax, semantics, and pragmatics.

24.2 Testing OOA and OOD Models

Analysis and design models cannot be tested in the conventional sense, because they cannot be executed. However, technical reviews (Chapter 20) can be used to examine their correctness and consistency.

24.2.1 Correctness of OOA and OOD Models

The notation and syntax used to represent analysis and design models will be tied to the specific analysis and design methods that are chosen for the project. Hence syntactic correctness is judged on proper use of the symbology; each model is reviewed to ensure that proper modeling conventions have been maintained.

During analysis and design, you can assess semantic correctness based on the model's conformance to the real-world problem domain. If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed), then it is semantically correct. To determine whether the model does, in fact, reflect real-world requirements, it should be presented to

problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity. Class relationships (instance connections) are evaluated to determine whether they accurately reflect real-world object connections.[2]

24.2.2 Consistency of Object-Oriented Models

The consistency of object-oriented models may be judged by "considering the relationships among entities in the model. An inconsistent analysis or design model has representations in one part that are not correctly reflected in other portions of the model" [McG94].

To assess consistency, you should examine each class and its connections to other classes. The class-responsibility-collaborator (CRC) model and an object-relationship diagram can be used to facilitate this activity. As you learned in Chapter 10, the CRC model is composed on CRC index cards. Each CRC card lists the class name, its responsibilities (operations), and its collaborators (other classes to which it sends messages and on which it depends for the accomplishment of its responsibilities). The collaborations imply a series of relationships (i.e., connections) between classes of the OO system. The object relationship model provides a graphic representation of the connections between classes. All of this information can be obtained from the analysis model (Chapter 10).

To evaluate the class model the following steps have been recommended [McG94]:

1. **Revisit the CRC model and the object-relationship model.** Cross-check to ensure that all collaborations implied by the requirements model are properly reflected in the both.
2. **Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.** For example, consider a class defined for a point-of-sale checkout system and called **CreditSale.** This class has a CRC index card as illustrated in Figure 24.1.

   For this collection of classes and collaborations, ask whether a responsibility (e.g., *read credit card*) is accomplished if delegated to the named collaborator (**CreditCard**). That is, does the class **CreditCard** have an operation that enables it to be read? In this case the answer is "yes." The object-relationship is traversed to ensure that all such connections are valid.
3. **Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.** For example, if the **CreditCard** class receives a request for **purchase amount** from the **CreditSale** class, there would be a problem. **CreditCard** does not know the purchase amount.

2 Use cases can be invaluable in tracking analysis and design models against real-world usage scenarios for the OO system.

FIGURE 24.1

An example CRC Index card used for review

class name: credit sale	
class type: transaction event	
class characteristics: nontangible, atomic, sequential, permanent, guarded	
responsibilities:	collaborators:
read credit card	credit card
get authorization	credit authority
post purchase amount	product ticket
	sales ledger
	audit file
generate bill	bill

4. **Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.**
5. **Determine whether widely requested responsibilities might be combined into a single responsibility.** For example, *read credit card* and *get authorization* occur in every situation. They might be combined into a *validate credit request* responsibility that incorporates getting the credit card number and gaining authorization.

You should apply steps 1 through 5 iteratively to each class and through each evolution of the requirements model.

Once the design model (Chapters 12–18) is created, you should also conduct reviews of the system design and the object design. The system design depicts the overall product architecture, the subsystems that compose the product, the manner in which subsystems are allocated to processors, the allocation of classes to subsystems, and the design of the user interface. The object model presents the details of each class and the messaging activities that are necessary to implement collaborations between classes.

The system design is reviewed by examining the object-behavior model developed during object-oriented analysis and mapping required system behavior against the subsystems designed to accomplish this behavior. Concurrency and task allocation are also reviewed within the context of system behavior. The behavioral states of the system are evaluated to determine which exist concurrently. Use cases are used to exercise the user interface design.

The object model should be tested against the object-relationship network to ensure that all design objects contain the necessary attributes and operations to

implement the collaborations defined for each CRC index card. In addition, the detailed specification of operation details (i.e., the algorithms that implement the operations) is reviewed.

24.3 Object-Oriented Testing Strategies

As we noted in Chapter 23, the classical software testing strategy begins with "testing in the small" and works outward toward "testing in the large." Stated in the jargon of software testing (Chapter 23), you begin with *unit testing,* then progress toward *integration testing,* and culminate with *validation and system testing*. In conventional applications, unit testing focuses on the smallest compilable program unit—the subprogram (e.g., component, module, subroutine, procedure). Once each of these units has been testing individually, it is integrated into a program structure while a series of regression tests are run to uncover errors due to interfacing the modules and side effects that are caused by the addition of new units. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

24.3.1 Unit Testing in the OO Context

KEY POINT

The smallest testable "unit" in OO software is the class. Class testing is driven by the operations encapsulated by the class and the state behavior of the class.

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

We can no longer test a single operation in isolation (the conventional view of unit testing) but rather, as part of a class. To illustrate, consider a class hierarchy in which an operation X() is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation X(), but it is applied within the context of the private attributes and operations that have been defined for each subclass. Because the context in which operation X() is used varies in subtle ways, it is necessary to test operation X() in the context of each of the subclasses. This means that testing operation X() in a vacuum (the traditional unit-testing approach) is ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software.[3] Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flows across the module

3 Test-case design methods for OO classes are discussed in Sections 24.4 through 24.6.

interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

24.3.2 Integration Testing in the OO Context

Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [Ber93].

KEY POINT

Integration testing for OO software tests a set of classes that are required to respond to a given event.

There are two different strategies for integration testing of OO systems [Bin94a]. The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed. Unlike conventional integration, the use of driver and stubs (Chapter 23) as replacement operations is to be avoided, when possible.

Cluster testing [McG94] is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

24.3.3 Validation Testing in an OO Context

At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable outputs from the system. To assist in the derivation of validation tests, the tester should draw upon use cases (Chapters 9 and 10) that are part of the requirements model. The use case provides a scenario that has a high likelihood of uncovered errors in user-interaction requirements.

Conventional black-box testing methods (Chapter 23) can be used to drive validation tests. In addition, you may choose to derive test cases from the object-behavior model and from an event flow diagram created as part of OOA.

24.4 Object-Oriented Testing Methods

The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) performs functions that help to achieve system

Quote:

"I see testers as the bodyguards of the project. We defend our developer's flank from failure, while they focus on creating success."

James Bach

requirements. It is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Test-case design methods for object-oriented software continue to evolve. However, an overall approach to OO test-case design has been suggested by Berard [Ber93]:

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain: a list of specified states for the class that is to be tested, a list of messages and operations that will be exercised as a consequence of the test, a list of exceptions that may occur as the class is tested, a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test), and supplementary information that will aid in understanding or implementing the test.

Unlike conventional test-case design, which is driven by an input-process-output view of software or the algorithmic detail of individual modules, object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

24.4.1 The Test-Case Design Implications of OO Concepts

As a class evolves through the analysis and design models, it becomes a target for test-case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder [Bin94a] notes, "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

WebRef

An excellent collection of papers and resources on OO testing can be found by searching the site **https://www.thecsiac.com/.**

Inheritance may also present you with additional challenges during test-case design. We have already noted that each new usage context requires retesting, even though reuse has been achieved. In addition, multiple inheritance[4] complicates testing further by increasing the number of contexts for which testing is required [Bin94a]. If subclasses instantiated from a superclass are used within the same problem domain, it is likely that the set of test cases derived for the superclass can be used when testing the subclass. However, if the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

4 An OO concept that should be used with extreme care.

24.4.2 Applicability of Conventional Test-Case Design Methods

The white-box testing methods described in Chapter 23 can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. As we noted in Chapter 23, use cases can provide useful input in the design of black-box and state-based tests.

24.4.3 Fault-Based Testing[5]

KEY POINT

The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.

The object of *fault-based testing* within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, preliminary planning required to perform fault-based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Of course, the effectiveness of these techniques depends on how testers perceive a plausible fault. If real faults in an OO system are perceived to be implausible, then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

? What types of faults are encountered in operation calls and message connections?

Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, and incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called

5 Sections 24.4.3 and 24.4.4 have been adapted from an article by Brian Marick originally posted on the Internet newsgroup comp.testing. This adaptation is included with the permission of the author. For further information on these topics, see [Mar94]. It should be noted that the techniques discussed in Sections 24.4.3 and 24.4.4 are also applicable for conventional software.

code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

24.4.4 Scenario-Based Test Design

Fault-based testing misses two main types of errors: (1) incorrect specifications and (2) interactions among subsystems. When errors associated with an incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong thing or omit important functionality. But in either circumstance, quality (conformance to requirements) suffers. Errors associated with subsystem interaction occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

KEY POINT

Scenario-based testing will uncover errors that occur when any actor interacts with the software.

Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests.

Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

24.5 Testing Methods Applicable at the Class Level

Testing "in the small" focuses on a single class and the methods that are encapsulated by the class. Random testing and partitioning are methods that can be used to exercise a class during OO testing.

24.5.1 Random Testing for OO Classes

ADVICE

The number of possible permutations for random testing can grow quite large. A strategy similar to orthogonal array testing can be used to improve testing efficiency.

To provide brief illustrations of these methods, consider a banking application in which an **Account** class has the following operations: *open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(),* and *close()* [Kir94]. Each of these operations may be applied for **Account,** but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of **Account** includes the following operations:

open•setup•deposit•withdraw•close

This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:

open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]^n•withdraw•close

A variety of different operation sequences can be generated randomly. For example:

Test case r_1: **open•setup•deposit•deposit•balance•summarize•withdraw•close**

Test case r_2: **open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close**

These and other random order tests are conducted to exercise different class instance life histories.

SafeHome

Class Testing

The scene: Shakira's cubicle.

The players: Jamie and Shakira—members of the *SafeHome* software engineering team who are working on test-case design for the security function.

The conversation:

Shakira: I've developed some tests for the **Detector** class [Figure 14.4]—you know, the one that allows access to all of the **Sensor** objects for the security function. You familiar with it?

Jamie (laughing): Sure, it's the one that allowed you to add the "doggie angst" sensor.

Shakira: The one and only. Anyway, it has an interface with four ops: *read(), enable(), disable(),*and *test().* Before a sensor can be read, it must be enabled. Once it's enabled, it can be read and tested. It can be disabled at any time, except if an alarm condition is being processed. So I defined a simple test sequence that will exercise its behavioral life history. [Shows Jamie the following sequence.]

#1: enable•test•read•disable

Jamie: That'll work, but you've got to do more testing than that!

Shakira: I know, I know, here are some other sequences I've come up with. [Shows Jamie the following sequences.]

#2: enable•test*[read]^n•test•disable

#3: [read]^n

#4: enable*disable•[test | read]

Jamie: So let me see if I understand the intent of these. #1 goes through a normal life history, sort of a conventional usage. #2 repeats the read operation *n* times, and that's a likely scenario. #3 tries to read the sensor before it's been enabled . . . that should produce an error message of some kind, right? #4 enables and disables the sensor and then tries to read it. Isn't that the same as test #2?

Shakira: Actually no. In #4, the sensor has been enabled. What #4 really tests is whether the disable op works as it should. A *read()* or *test()* after *disable()* should generate the error message. If it doesn't, then we have an error in the disable op.

Jamie: Cool. Just remember that the four tests have to be applied for every sensor type since all the ops may be subtly different depending on the type of sensor.

Shakira: Not to worry. That's the plan.

24.5.2 Partition Testing at the Class Level

? What testing options are available at the class level?

Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning (Chapter 23) for traditional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

Quote:

"The boundary that defines the scope of unit and integration testing is different for object-oriented development. Tests can be designed and exercised at many points in the process. Thus 'design a little, code a little' becomes 'design a little, code a little, test a little.'

Robert Binder

State-based partitioning categorizes class operations based on their ability to change the state of the class. As an example, consider the **Account** class. State operations include *deposit()* and *withdraw(),* whereas nonstate operations include *balance(), summarize(),* and *creditLimit().*Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,

Test case p_1: **open•setup•deposit•deposit•withdraw•withdraw•close**

Test case p_2: **open•setup•deposit•summarize•creditLimit•withdraw•close**

Test case p_1 changes state, while test case $p_2$exercises operations that do not change state (other than those in the minimum test sequence).

Other types of partition testing can also be applied. *Attribute-based partitioning* categorizes class operations based on the attributes that they use. *Category-based partitioning* categorizes class operations based on the generic function that each performs.

24.6 Interclass Test-Case Design

Test-case design becomes more complicated as integration of the object-oriented system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test-case generation" [Kir94], we expand the banking example introduced in Section 24.5 to include the classes and collaborations noted in Figure 24.2. The direction of the arrows in the figure indicates the direction of messages, and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

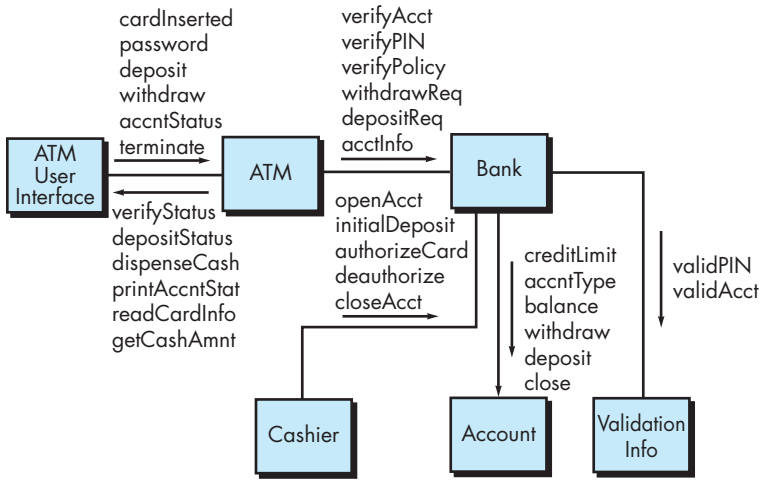
24.6.1 Multiple Class Testing

Kirani and Tsai [Kir94] suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.

FIGURE 24.2

Class collaboration diagram for banking application. *Source:* Adapted from [Kir94].



3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate [Kir94], consider a sequence of operations for the **Bank** class relative to an **ATM** class (Figure 24.2):

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]^n

A random test case for the **Bank** class might be

Test case r_3 = **verifyAcct•verifyPIN•depositReq**

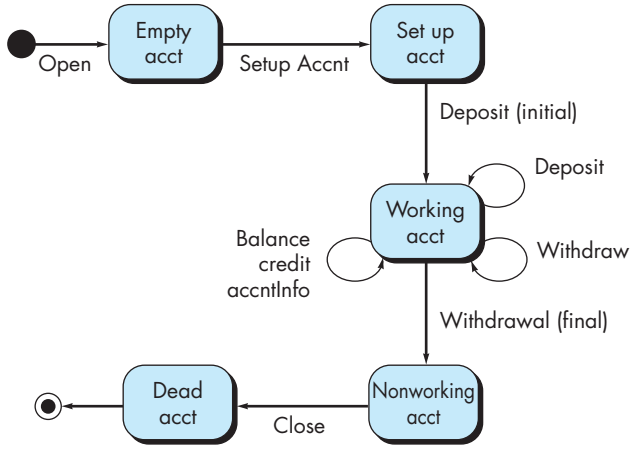
In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case r_3 are considered. **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct()* and *verifyPIN()*. **Bank** must collaborate with **Account** to execute *depositReq()*. Hence, a new test case that exercises these collaborations is

Test case r_4 = **verifyAcct [Bank:validAcctValidationInfo]•verifyPIN [Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]**

The approach for multiple class partition testing is similar to the approach used for partition testing of individual classes. A single class is partitioned as discussed in Section 24.5.2. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative

FIGURE 24.3

State diagram for the Account class. *Source:* Adapted from [Kir94].



approach partitions tests based on the interfaces to a particular class. Referring to Figure 24.2, the **Bank** class receives messages from the **ATM** and **Cashier** classes. The methods within **Bank** can therefore be tested by partitioning them into those that serve **ATM** and those that serve **Cashier.** State-based partitioning (Section 24.5.2) can be used to refine the partitions further.

24.6.2 Tests Derived from Behavior Models

The use of the state diagram as a model that represents the dynamic behavior of a class is discussed in Chapter 11. The state diagram for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure 24.3 [Kir94] illustrates a state diagram for the **Account** class discussed earlier. Referring to the figure, initial transitions move through the *empty acct* and *setup acct* states. The majority of all behavior for instances of the class occurs while in the *working acct* state. A final withdrawal and account closure cause the account class to make transitions to the *nonworking acct* and *dead acct* states, respectively.

The tests to be designed should achieve coverage of every state. That is, the operation sequences should cause the **Account** class to make transition through all allowable states:

Test case s_1: **open•setupAccnt•deposit (initial)•withdraw (final)•close**

It should be noted that this sequence is identical to the minimum test sequence discussed in Section 24.5.2. Adding additional test sequences to the minimum sequence,

Test case s_2: **open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close**

Test case s_3: **open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close**

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple state diagrams are used to track the behavioral flow of the system.

The state model can be traversed in a "breadth-first" [McG94] manner. In this context, breadth-first implies that a test case exercises a single transition and that when a new transition is to be tested, only previously tested transitions are used.

Consider a **CreditCard** object that is part of the banking system. The initial state of **CreditCard** is *undefined* (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a *defined* state; that is, the attributes **card number** and **expiration date**, along with bank-specific identifiers are defined. The credit card is *submitted* when it is sent for authorization, and it is *approved* when authorization is received. The transition of **CreditCard** from one state to another can be tested by deriving test cases that cause the transition to occur. A breadth-first approach to this type of testing would not exercise *submitted* before it exercised *undefined* and *defined.*If it did, it would make use of transitions that had not been previously tested and would therefore violate the breadth-first criterion.

24.7 Summary

The overall objective of object-oriented testing—to find the maximum number of errors with a minimum amount of effort is identical to the objective of conventional software testing. But the strategy and tactics for OO testing differ significantly. The view of testing broadens to include the review of both the requirements and design model. In addition, the focus of testing moves away from the procedural component (the module) and toward the class.

Because the OO requirements and design models and the resulting source code are semantically coupled, testing (in the form of technical reviews) begins during the modeling activity. For this reason, the review of CRC, object-relationship, and object-behavior models can be viewed as first-stage testing.

Once code is available, unit testing is applied for each class. The design of tests for a class uses a variety of methods: fault-based testing, random testing, and partition testing. Each of these methods exercise the operations encapsulated by the class. Test sequences are designed to ensure that relevant operations are exercised. The state of the class, represented by the values of its attributes, is examined to determine if errors exist.

Integration testing can be accomplished using a thread-based or use-based strategy. Thread-based testing integrates the set of classes that collaborate to respond to one input or event. Use-based testing constructs the system in layers, beginning with those classes that do not make use of server classes. Integration

test-case design methods can also make use of random and partition tests. In addition, scenario-based testing and tests derived from behavioral models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations.

OO system validation testing is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software. However, scenario-based testing dominates the validation of OO systems, making the use case a primary driver for validation testing.

Problems and Points to Ponder

24.1. In your own words, describe why the class is the smallest reasonable unit for testing within an OO system.

24.2. Why do we have to retest subclasses that are instantiated from an existing class, if the existing class has already been thoroughly tested? Can we use the test-case design for the existing class?

24.3. Why should "testing" begin with object-oriented analysis and design?

24.4. Derive a set of CRC index cards for *SafeHome,* and conduct the steps noted in Section 24.2.2 to determine if inconsistencies exist.

24.5. What is the difference between thread-based and use-based strategies for integration testing? How does cluster testing fit in?

24.6. Apply random testing and partitioning to three classes defined in the design for the *SafeHome* system. Produce test cases that indicate the operation sequences that will be invoked.

24.7. Apply multiple class testing and tests derived from the behavioral model for the *SafeHome* design.

24.8. Derive four additional tests using random testing and partitioning methods as well as multiple class testing and tests derived from the behavioral model for the banking application presented in Sections 24.5 and 24.6.

Further Readings and Information Sources

Many books on testing noted in the *Further Readings* sections of Chapters 22 and 23 discuss testing of OO systems to some extent. Bashir and Goel (*Testing Object-Oriented Software,* Springer, 2012), Schach (*Object-Oriented and Classical Software Engineering,* McGraw-Hill, 8th ed., 2010) and Bruege and Dutoit (*Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 3rd ed., 2009) consider OO testing within the context of broader software engineering practice. Jorgensen (*Software Testing: A Craftsman's Approach,* Auerbach, 3rd ed., 2008) discusses formal techniques and object-oriented techniques. Yurga (*Testing and Testability of Object-Oriented Software Systems via Metrics: A Metrics-Based Approach to the Testing Process and Testability of Object-Oriented Software Systems,* LAP Lambert, 2011), Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), and Kung and his colleagues (*Testing Object-Oriented Software,* Wiley-IEEE Computer Society Press, 1998), treat OO testing in significant detail. Freeman

and Pryce (*Growing Object-Oriented Software, Guided by Tests,* Addison-Wesley, 2009) discussing test-driven design of object-oriented software. Denney (*Use Case Levels of Test: A Four-Step Strategy for Building Time and Innovation in Software Test Design,* CreateSpace Independent Publishing, 2012) discusses techniques that may be applied to applied testing object-oriented systems.

A wide variety of information sources on object-oriented testing methods are available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing techniques can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

25 Testing Web Applications

Key Concepts

compatibility tests 554
component-level testing 555
configuration testing 558
content testing ... 545
database testing .. 547
load testing 562
navigation testing . 556
performance testing 560
planning 543
quality, dimensions of 541
security testing ... 559
strategy 543
stress testing 562
usability tests 552
user interface testing 549

There is an urgency that always pervades a WebApp project. Stakeholders—concerned about competition from other WebApps, coerced by customer demands, and worried that they'll miss a market window—press to get the WebApp online. As a consequence, technical activities that often occur late in the process, such as WebApp testing, are sometimes given short shrift. This can be a catastrophic mistake. To avoid it, you and other team members must ensure that each work product exhibits high quality. Wallace and his colleagues [Wal03] note this when they state:

> Testing shouldn't wait until the project is finished. Start testing before you write one line of code. Test constantly and effectively, and you will develop a much more durable Web site.

Since requirements and design models cannot be tested in the classical sense, you and your team should conduct technical reviews (Chapter 20) as well as executable tests. The intent is to uncover and correct errors before the WebApp is made available to its end users.

Quick Look

What is it? WebApp testing is a collection of related activities with a single goal: to uncover errors in WebApp content, function, usability, navigability, performance, capacity, and security. To accomplish this, a testing strategy that encompasses both reviews and executable testing is applied.

Who does it? Web engineers and other project stakeholders (managers, customers, end users) all participate in WebApp testing.

Why is it important? If end users encounter errors that shake their faith in the WebApp, they will go elsewhere for the content and function they need, and the WebApp will fail. For this reason, you must work to eliminate as many errors as possible before the WebApp goes online.

What are the steps? The WebApp testing process begins by focusing on user-visible aspects of the WebApp and proceeds to tests that exercise technology and infrastructure. Seven testing steps are performed: content testing, interface testing, navigation testing, component testing, configuration testing, performance testing, and security testing.

What is the work product? In some instances a WebApp test plan is produced. In every instance, a suite of test cases is developed for every testing step and an archive of test results is maintained for future use.

How do I ensure that I've done it right? Although you can never be sure that you've performed every test that is needed, you can be certain that testing has uncovered errors (and that those errors have been corrected). In addition, if you've established a test plan, you can check to ensure that all planned tests have been conducted.

25.1 Testing Concepts for WebApps

Testing is the process of exercising software with the intent of finding (and ultimately correcting) errors. This fundamental philosophy, first presented in Chapter 22, does not change for WebApps. In fact, because Web-based systems and applications reside on a network and interoperate with many different operating systems, browsers (or other personal communication devices), hardware platforms, communications protocols, and "backroom" applications, the search for errors represents a significant challenge.

To understand the objectives of testing within a Web engineering context, you should consider the many dimensions of WebApp quality.[1] In the context of this discussion, we consider quality dimensions that are particularly relevant in any discussion of WebApp testing. We also consider the nature of the errors that are encountered as a consequence of testing, and the testing strategy that is applied to uncover these errors.

25.1.1 Dimensions of Quality

Quality is incorporated into a Web application as a consequence of good design. It is evaluated by applying a series of technical reviews that assess various elements of the design model and by applying a testing process that is discussed throughout this chapter. Both reviews and testing examine one or more of the following quality dimensions [Mil00a]:

? How do we assess quality within the context of a WebApp and its environment?

- *Content* is evaluated at both a syntactic and semantic level. At the syntactic level, spelling, punctuation, and grammar are assessed for text-based documents. At a semantic level, correctness (of information presented), consistency (across the entire content object and related objects), and lack of ambiguity are all assessed.
- *Function* is tested to uncover errors that indicate lack of conformance to customer requirements. Each WebApp function is assessed for correctness, instability, and general conformance to appropriate implementation standards (e.g., Java or AJAX standards).
- *Structure* is assessed to ensure that it properly delivers WebApp content and function, that it is extensible, and that it can be supported as new content or functionality is added.
- *Usability* is tested to ensure that each category of user is supported by the interface and can learn and apply all required navigation syntax and semantics.

1 Generic software quality dimensions, equally applicable for WebApps, were discussed in Chapter 19.

Quote:

"Innovation is a bittersweet deal for software testers. Just when it seems that we know how to test a particular technology, a new one [WebApps] comes along and all bets are off."

James Bach

- *Navigability* is tested to ensure that all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
- *Performance* is tested under a variety of operating conditions, configurations, and loading to ensure that the system is responsive to user interaction and handles extreme loading without unacceptable operational degradation.
- *Compatibility* is tested by executing the WebApp in a variety of different host configurations on both the client and server sides. The intent is to find errors that are specific to a unique host configuration.
- *Interoperability*is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- *Security* is tested by assessing potential vulnerabilities and attempting to exploit each. Any successful penetration attempt is deemed a security failure.

A strategy and tactics for WebApp testing has been developed to exercise each of these quality dimensions and is discussed in the remainder of this chapter.

25.1.2 Errors within a WebApp Environment

Errors encountered as a consequence of successful WebApp testing have a number of unique characteristics [Ngu00]:

? What makes errors encountered during WebApp execution somewhat different from those encountered for conventional software?

1. Because many types of WebApp tests uncover problems that are first evidenced on the client side (i.e., via an interface implemented on a specific browser or a personal communication device), you often see a symptom of the error, not the error itself.
2. Because a WebApp is implemented in a number of different configurations and within different environments, it may be difficult or impossible to reproduce an error outside the environment in which the error was originally encountered.
3. Although some errors are the result of incorrect design or improper HTML (or other programming language) coding, many errors can be traced to the WebApp configuration.
4. Because WebApps reside within a client-server architecture, errors can be difficult to trace across three architectural layers: the client, the server, or the network itself.
5. Some errors are due to the *static operating environment* (i.e., the specific configuration in which testing is conducted), while others are attributable to the dynamic operating environment (i.e., instantaneous resource loading or time-related errors).

These five error attributes suggest that environment plays an important role in the diagnosis of all errors uncovered during the WebApp testing. In some situations (e.g., content testing), the site of the error is obvious, but in many other types of WebApp testing (e.g., navigation testing, performance testing, security testing) the underlying cause of the error may be considerably more difficult to determine.

25.1.3 Testing Strategy

The strategy for WebApp testing adopts the basic principles for all software testing (Chapter 22) and applies a strategy and tactics that have been recommended for object-oriented systems (Chapter 24). The following steps summarize the approach:

KEY POINT
The overall strategy for WebApp testing can be summarized in the 10 steps noted here.

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Selected functional components are unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users; the results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp security, reliability, and performance.

WebRef
Excellent articles on WebApp testing can be found at **www.stickyminds.com/testing.asp.**

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by Web support staff who use regression tests derived from the tests developed when the WebApp was first engineered.

25.1.4 Test Planning

The use of the word *planning* (in any context) is anathema to some Web developers. These developers don't plan; they just start—hoping that a killer WebApp

will emerge. A more disciplined approach recognizes that planning establishes a road map for all work that follows. It's worth the effort. In their book on WebApp testing, Splaine and Jaskiel [Spl01] state:

KEY POINT

The test plan identifies the testing task set, the work products to be developed, and the way in which results are to be evaluated, recorded, and reused.

> Except for the simplest of websites, it quickly becomes apparent that some sort of test planning is needed. All too often, the initial number of bugs found from ad hoc testing is large enough that not all of them are fixed the first time they're detected. This puts an additional burden on people who test websites and applications. Not only must they conjure up imaginative new tests, but they must also remember how previous tests were executed in order to reliably re-test the website/application, and ensure that known bugs have been removed and that no new bugs have been introduced.

The questions you should ask are: How do we "conjure up imaginative new tests," and what should those tests focus on? The answers to these questions are contained within a test plan.

A WebApp test plan identifies (1) the task set[2] to be applied as testing commences, (2) the work products to be produced as each testing task is executed, and (3) the manner in which the results of testing are evaluated, recorded, and reused when regression testing is conducted. In some cases, the test plan is integrated with the project plan. In others, the test plan is a separate document.

25.2 The Testing Process—An Overview

Quote:

"In general, the software testing techniques that are applied to other applications are the same as those applied to Web-based applications . . . The difference between the two types of testing is that the technology variables in the Web environment multiply."

Hung Nguyen

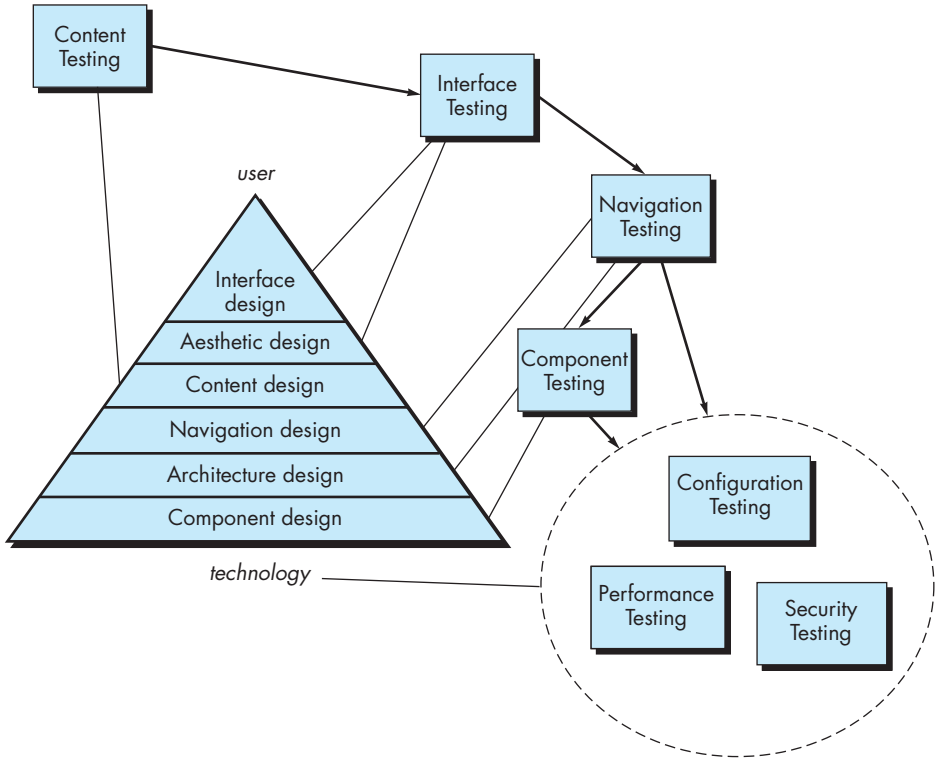
You begin the WebApp testing process with tests that exercise content and interface functionality that are immediately visible to end users. As testing proceeds, aspects of the design architecture and navigation are exercised. Finally, the focus shifts to tests that examine technological capabilities that are not always apparent to end users—WebApp infrastructure and installation/implementation issues.

Figure 25.1 juxtaposes the WebApp testing process with the design pyramid for WebApps (Chapter 17). Note that as the testing flow proceeds from left to right and top to bottom, user-visible elements of the WebApp design (top elements of the pyramid) are tested first, followed by infrastructure design elements.

2 Task sets are discussed in Chapter 3. A related term–*workflow*">–is also used to describe a series of tasks required to accomplish a software engineering activity.

FIGURE 25.1

The testing process



25.3 CONTENT TESTING

ADVICE

Although technical reviews are not a part of testing, content review should be performed to ensure that content has quality.

Errors in WebApp content can be as trivial as minor typographical mistakes or as significant as incorrect information, improper organization, or violation of intellectual property laws. *Content testing* attempts to uncover these and many other problems before the user encounters them.

Content testing combines both reviews and the generation of executable test cases. Review is applied to uncover semantic errors in content (discussed in Section 25.3.1). Executable testing is used to uncover content errors that can be traced to dynamically derived content that is driven by data acquired from one or more databases.

KEY POINT

Content testing objects are: (1) to uncover syntactic errors in content, (2) to uncover semantic errors, and (3) to find structural errors.

25.3.1 Content Testing Objectives

Content testing has three important objectives: (1) to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media; (2) to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs, and (3) to find errors in the organization or structure of content that is presented to the end user.

To accomplish the first objective, automated spelling and grammar checkers may be used. However, many syntactic errors evade detection by such tools and must be discovered by a human reviewer (tester). In fact, a large website might enlist the services of a professional copy editor to uncover typographical errors, grammatical mistakes, errors in content consistency, errors in graphical representations, and cross-referencing errors.

Semantic testing focuses on the information presented within each content object. The reviewer (tester) must answer the following questions:

? What questions should be asked and answered to uncover semantic errors in content?

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Obtaining answers to each of these questions for a large WebApp (containing hundreds of content objects) can be a daunting task. However, failure to uncover semantic errors will shake the user's faith in the WebApp and can lead to failure of the Web-based application.

Content objects exist within an architecture that has a specific style (Chapter 17). During content testing, the structure and organization of the content architecture is tested to ensure that required content is presented to the end user in the proper order and relationships. For example, the **SafeHomeAssured.com** WebApp presents a variety of information about sensors that are used as part of security and surveillance products. Content objects provide descriptive information, technical specifications, a photographic representation, and related information. Tests of the **SafeHomeAssured.com** content architecture strive to uncover errors in the presentation of this information (e.g., a description of Sensor *X* is presented with a photo of Sensor *Y*).

INFO

Web Content Testing Tools

Objective: The objective of Web content testing tools is to identify errors that prevent a Web page from displaying web content in a readable and organized manner.

Mechanics: These tools typically prompt you to enter the URL of a web resource to test. Each tool then provides a list of errors (e.g., failure to follow the markup language standard) with suggestions on how to correct the errors.

Representative Tools:[3]

http://validator.w3.org/—Online WC3 tool checks Web pages for markup language validity (HTM, XHTML, SMIL, MathML).

http://jigsaw.w3.org/css-validator/—Online WC3 tool that checks CSS style sheets and documents using CSS stylesheets.

http://validator.w3.org/feed/—Online WC3 tool that checks syntax of Atom or RSS feeds.

25.3.2 Database Testing

Modern WebApps do much more than present static content objects. In many application domains, WebApps interface with sophisticated database management systems and build dynamic content objects that are created in real time using the data acquired from a database.

For example, a financial services WebApp can produce complex text-based, tabular, and graphical information about a specific equity (e.g., a stock or mutual fund). The composite content object that presents this information is created dynamically after the user has made a request for information about a specific equity. To accomplish this, the following steps are required: (1) an equities database is queried, (2) relevant data are extracted from the database, (3) the extracted data must be organized as a content object, and (4) this content object (representing customized information requested by an end user) is transmitted to the client environment for display. Errors can and do occur as a consequence of each of these steps. The objective of database testing is to uncover these errors, but database testing is complicated by a variety of factors:

? What issues complicate database testing for WebApps?

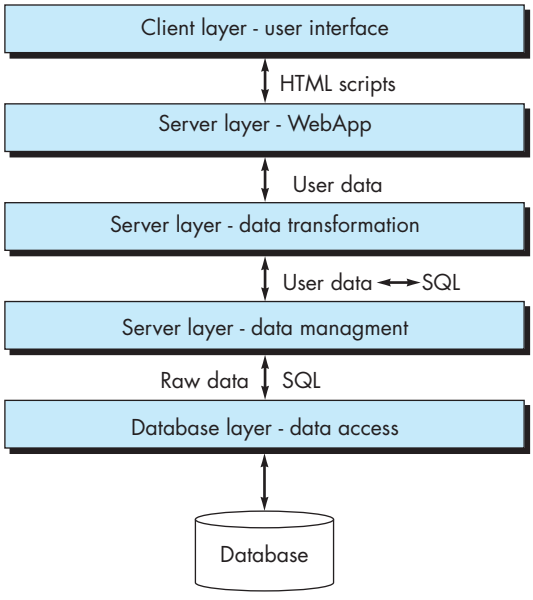
1. *The original client-side request for information is rarely presented in the form [e.g., structured query language (SQL)] that can be input to a database management system (DBMS).* Therefore, tests should be designed to uncover errors made in translating the user's request into a form that can be processed by these DBMS.
2. *The database may be remote to the server that houses the WebApp.* Therefore, tests that uncover errors in communication between the WebApp and the remote database must be developed.[4]

3 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

4 These tests can become complex when distributed databases are encountered or when access to a data warehouse (Chapter 1) is required.

FIGURE 25.2

Layers of interaction



Quote:

"[W]e are unlikely to have confidence in a website that suffers frequent downtime, hangs in the middle of a transaction, or has a poor sense of usability. Testing, therefore, has a crucial role in the overall development process."

Wing Lam

3. *Raw data acquired from the database must be transmitted to the WebApp server and properly formatted for subsequent transmittal to the client.* Therefore, tests that demonstrate the validity of the raw data received by the WebApp server must be developed, and additional tests that demonstrate the validity of the transformations applied to the raw data to create valid content objects must also be created.
4. *The dynamic content object(s) must be transmitted to the client in a form that can be displayed to the end user.* Therefore, a series of tests must be designed to (1) uncover errors in the content object format and (2) test compatibility with different client environment configurations.

Considering these four factors, test-case design methods should be applied for each of the "layers of interaction" [Ngu01] noted in Figure 25.2. Testing should ensure that (1) valid information is passed between the client and server from the interface layer, (2) the WebApp processes script correctly and properly extract or format user data, (3) user data are passed correctly to a server-side data transformation function that formats appropriate queries (e.g., SQL), (4) queries are passed to a data management layer[5] that communicates with database access routines (potentially located on another machine).

The data transformation, data management, and database access layers shown in Figure 25.2 are often constructed with reusable components that have

5 The data management layer typically incorporates an SQL call-level interface (SQL-CLI) such as Microsoft OLE/ADO or Java Database Connectivity (JDBC).

been validated separately and as a package. If this is the case, WebApp testing focuses on the design of test cases to exercise the interactions between the client layer and the first two server layers (WebApp and data transformation) shown in the figure.

The user interface layer is tested to ensure that scripts are properly constructed for each user query and properly transmitted to the server side. The WebApp layer on the server side is tested to ensure that user data are properly extracted from scripts and properly transmitted to the data transformation layer on the server side. The data transformation functions are tested to ensure that the correct SQL is created and passed to appropriate data management components.

A detailed discussion of the underlying technology that must be understood to adequately design these database tests is beyond the scope of this book. If you have additional interest, see [Sce02], [Ngu01], and [Bro01].

25.4 User Interface Testing

ADVICE

With the exception of WebApp-oriented specifics, the interface strategy noted here is applicable to all types of client-server software.

Verification and validation of a WebApp user interface occurs at three distinct points. During requirements analysis, the interface model is reviewed to ensure that it conforms to stakeholder requirements and to other elements of the requirements model. During design the interface design model is reviewed to ensure that generic quality criteria established for all user interfaces (Chapter 15) have been achieved and that application-specific interface design issues have been properly addressed. During testing, the focus shifts to the execution of application-specific aspects of user interaction as they are manifested by interface syntax and semantics. In addition, testing provides a final assessment of usability.

25.4.1 Interface Testing Strategy

Interface testing exercises interaction mechanisms and validates aesthetic aspects of the user interface. The overall strategy for interface testing is to (1) uncover errors related to specific interface mechanisms (e.g., errors in the proper execution of a menu link or the way data are entered in a form) and (2) uncover errors in the way the interface implements the semantics of navigation, WebApp functionality, or content display. To accomplish this strategy, a number of tactical steps are initiated:

- *Interface features are tested to ensure that design rules, aesthetics, and related visual content are available for the user without error.*
- *Individual interface mechanisms are tested in a manner that is analogous to unit testing.* For example, tests are designed to exercise all forms, client-side scripting, dynamic HTML, scripts, streaming content, and

application-specific interface mechanisms (e.g., a shopping cart for an e-commerce application).

- *Each interface mechanism is tested within the context of a use case or NSU (Chapter 17) for a specific user category*
- *The complete interface is tested against selected use cases and NSUs to uncover errors in the semantics of the interface.* It is at this stage that a series of usability tests are conducted.
- *The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible*

25.4.2 Testing Interface Mechanisms

When a user interacts with a WebApp, the interaction occurs through one or more interface mechanisms. A brief overview of testing considerations for each interface mechanism is presented in the paragraphs that follow [Spl01].

ADVICE

External link testing should occur throughout the life of the WebApp. Part of a support strategy should be regularly scheduled link tests.

Links. Each navigation link is tested to ensure that the proper content object or function is reached.[6] Testing includes links associated with the interface layout (e.g., menu bars, index items, links within each content object, and links to external WebApps).

Forms. At a macroscopic level, tests are performed to ensure that (1) labels correctly identify fields within the form and that mandatory fields are identified visually for the user, (2) the server receives all information contained within the form and that no data are lost in the transmission between client and server, (3) appropriate defaults are used when the user does not select from a pull-down menu or set of buttons, (4) browser functions (e.g., the "back" arrow) do not corrupt data entered in a form, and (5) scripts that perform error checking on data entered work properly and provide meaningful error messages.

At a more targeted level, tests should ensure that (1) form fields have proper width and data types, (2) the form establishes appropriate safeguards that preclude the user from entering text strings longer than some predefined maximum, (3) all appropriate options for pull-down menus are specified and ordered in a way that is meaningful to the end user, (4) browser "auto-fill" features do not lead to data input errors, and (5) tab key (or some other key) initiates proper movement between form fields.

ADVICE

Client-side scripting tests and tests associated with dynamic HTML should be repeated whenever a new version of a popular browser is released.

Client-side scripting. Black-box tests are conducted to uncover any errors in processing as the script is executed. These tests are often coupled with forms testing, because script input is often derived from data provided as part of forms processing.

6 These tests can be performed as part of either interface or navigation testing.

Dynamic HTML. Each Web page that contains dynamic HTML is executed to ensure that the dynamic display is correct. In addition, a compatibility test should be conducted to ensure that the dynamic HTML works properly in the environmental configurations that support the WebApp.

Pop-up windows. A series of tests ensure that (1) the pop-up is properly sized and positioned, (2) the pop-up does not cover the original WebApp window, (3) the aesthetic design of the pop-up is consistent with the aesthetic design of the interface, and (4) scroll bars and other control mechanisms appended to the pop-up are properly located and function as required.

CGI scripts. Black-box tests are conducted with an emphasis on data integrity (as data are passed to the CGI script) and script processing (once validated data have been received). In addition, performance testing can be conducted to ensure that the server-side configuration can accommodate the processing demands of multiple invocations of CGI scripts [Spl01].

Streaming content. Tests should demonstrate that streaming data are up-to-date, properly displayed, and can be suspended without error and restarted without difficulty.

Cookies. Both server-side and client-side testing are required. On the server side, tests should ensure that a cookie is properly constructed (contains correct data) and properly transmitted to the client side when specific content or functionality is requested. In addition, the proper persistence of the cookie is tested to ensure that its expiration date is correct. On the client side, tests determine whether the WebApp properly attaches existing cookies to a specific request (sent to the server).

Application-specific interface mechanisms. Tests conform to a checklist of functionality and features that are defined by the interface mechanism. For example, Splaine and Jaskiel [Spl01] suggest the following checklist for shopping cart functionality defined for an e-commerce application:

- Boundary-test (Chapter 23) the minimum and maximum number of items that can be placed in the shopping cart.
- Test a "check out" request for an empty shopping cart.
- Test proper deletion of an item from the shopping cart.
- Test to determine whether a purchase empties the cart of its contents.
- Test to determine the persistence of shopping cart contents (this should be specified as part of customer requirements).
- Test to determine whether the WebApp can recall shopping cart contents at some future date (assuming that no purchase was made).

25.4.3 Testing Interface Semantics

Once each interface mechanism has been "unit" tested, the focus of interface testing changes to a consideration of interface semantics. Interface semantics testing "evaluates how well the design takes care of users, offers clear direction, delivers feedback, and maintains consistency of language and approach" [Ngu00].

A thorough review of the interface design model can provide partial answers to the questions implied by the preceding paragraph. However, each use-case scenario (for each user category) should be tested once the WebApp has been implemented. In essence, a use case becomes the input for the design of a testing sequence. The intent of the testing sequence is to uncover errors that will preclude a user from achieving the objective associated with the use case.

25.4.4 Usability Tests

WebRef

A worthwhile guidance to usability testing can be found at **http://www.keyrelevance.com/articles/usability-tips.htm.**

Usability testing is similar to interface semantics testing (Section 25.4.3) in the sense that it also evaluates the degree to which users can interact effectively with the WebApp and the degree to which the WebApp guides users' actions, provides meaningful feedback, and enforces a consistent interaction approach. Rather than focusing intently on the semantics of some interactive objective, usability reviews and tests are designed to determine the degree to which the WebApp interface makes the user's life easy.[7]

You will invariably contribute to the design of usability tests, but the tests themselves are conducted by end users. Usability testing can occur at a variety of different levels of abstraction: (1) the usability of a specific interface mechanism (e.g., a form) can be assessed, (2) the usability of a complete Web page (encompassing interface mechanisms, data objects, and related functions) can be evaluated, or (3) the usability of the complete WebApp can be considered.

? What characteristics of usability become the focus of testing and what specific objectives are addressed?

The first step in usability testing is to identify a set of usability categories and establish testing objectives for each category. The following test categories and objectives (written in the form of a question) illustrate this approach:[8]

Interactivity—Are interaction mechanisms (e.g., pull-down menus, buttons, pointers) easy to understand and use?

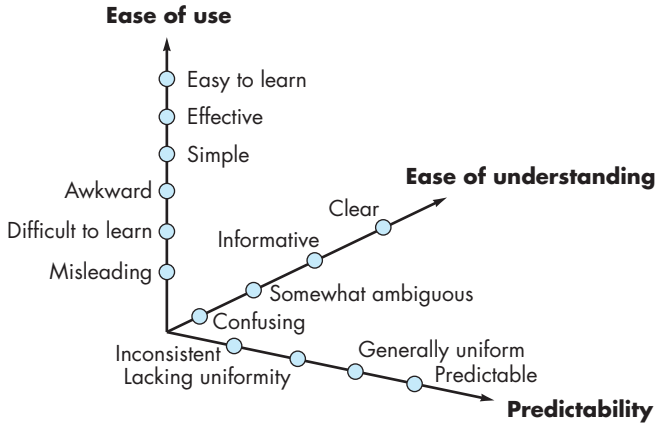
Layout—Are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?

7 The term *user-friendliness* has been used in this context. The problem, of course, is that one user's perception of a "friendly" interface may be radically different from another's.

8 For additional information on usability, see Chapter 15.

FIGURE 25.3

Qualitative assessment of usability



Readability—Is text well written and understandable?[9] Are graphic representations easy to understand?

Aesthetics—Do layout, color, typeface, and related characteristics lead to ease of use? Do users "feel comfortable" with the look and feel of the WebApp?

Display characteristics—Does the WebApp make optimal use of screen size and resolution?

Time sensitivity—Can important features, functions, and content be used or acquired in a timely manner?

Personalization—Does the WebApp tailor itself to the specific needs of different user categories or individual users?

Accessibility—Is the WebApp accessible to people who have disabilities?

A series of tests is designed within each of these categories. In some cases, the "test" may be a visual review of a Web page. In other cases interface semantics tests may be executed again, but in this instance usability concerns are paramount.

As an example, we consider usability assessment for interaction and interface mechanisms. Constantine and Lockwood [Con99] suggest that the following list of interface features should be reviewed and tested for usability: animation, buttons, color, control, dialogue, fields, forms, frames, graphics, labels, links, menus, messages, navigation, pages, selectors, text, and tool bars. As each feature is assessed, it is graded on a qualitative scale by the users who are doing the testing. Figure 25.3 depicts a possible set of assessment "grades" that can be selected by users. These grades are applied to each feature individually, to a complete Web page, or to the WebApp as a whole.

9 The FOG Readability Index and others may be used to provide a quantitative assessment of readability See **http://developer.gnome.org/gdp-style-guide/stable/usability-readability.html.en** for more details.

INFO

Web User Interface Testing Tools

Objective: The objective of Web user interface testing tools is to determine usability or accessibility problems present in a Web page or website.

Mechanics: Two types of tools are listed here. Some tools prompt you to enter the URL of a Web page and print a list of suggestions on how to correct any failures to conform to a set of heuristics. Some tools capture the user actions and prompt users for feedback while working on the pages of a website.

Representative Tools:[10]

http://www.usabilla.com/—*Usabilla* is an online tool that allows developers to track user actions and gather opinions during active use of a Web page.

http://www.google.com/analytics/—*Google Analytics* is an online tool that provides a comprehensive set of website data tracking and analysis tools that can be used to assess site usability.

http://valet.webthing.com/access/url.html—*Web Valet* provides an online service for checking Web pages for accessibility issues.

http://wave.webaim.org/—*Wave* provides an online service that marks up Web pages to show accessibility issues.

http://www.sidar.org/hera/index.php.en—*Hera* provides an online service that uses the Web Content Accessibility Guidelines to check Web pages for accessibility issues.

25.4.5 Compatibility Tests

Different computers, display devices, operating systems, browsers, and network connection speeds can have a significant influence on WebApp operation. Each computing configuration can result in differences in client-side processing speeds, display resolution, and connection speeds. Operating system vagaries may cause WebApp processing issues. Different browsers sometimes produce slightly different results, regardless of the degree of HTML standardization within the WebApp. Required plug-ins may or may not be readily available for a particular configuration. *Compatibility testing* strives to uncover these problems before the WebApp goes online.

The first step in compatibility testing is to define a set of "commonly encountered" client-side computing configurations and their variants. In essence, a tree structure is created, identifying each computing platform, typical display devices, the operating systems supported on the platform, the browsers available, likely Internet connection speeds, and similar information. Next, a series of compatibility validation tests are derived, often adapted from existing interface tests, navigation tests, performance tests, and security tests. The intent of these tests is to uncover errors or execution problems that can be traced to configuration differences.

10 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

SafeHome

WebApp Testing

The scene: Doug Miller's office.

The players: Doug Miller, manager of the *SafeHome* software engineering group, and Vinod Raman, a member of the product software engineering team.

The conversation:

Doug: What do you think of the **SafeHomeAssured.com** e-commerce WebApp V0.0?

Vinod: The outsourcing vendor has done a good job. Sharon (development manager for the vendor) tells me they're testing as we speak.

Doug: I'd like you and the rest of the team to do a little informal testing on the e-commerce site.

Vinod (grimacing): I thought we were going to hire a third-party testing company to validate the WebApp. We're still killing ourselves trying to get the product software out the door.

Doug: We're going to hire a testing vendor for performance and security testing, and our outsourcing vendor is already testing. Just thought another point of view would be helpful, and besides, we'd like to keep costs in line, so . . .

Vinod (sighs): What are you looking for?

Doug: I want to be sure that the interface and all navigation are solid.

Vinod: I suppose we can start with the use cases for each of the major interface functions:

Learn about *SafeHome*.

Specify the *SafeHome* system you need.

Purchase a *SafeHome* system.

Get technical support.

Doug: Good. But take the navigation paths all the way to their conclusion.

Vinod (looking through a notebook of use cases): Yeah, when you select **Specify the *SafeHome* system you need**, that'll take you to:

Select *SafeHome* components.

Get *SafeHome* component recommendations.

We can exercise the semantics of each path.

Doug: While you're there, check out the content that appears at each navigation node.

Vinod: Of course . . . and the functional elements as well. Who's testing usability?

Doug: Oh . . . the testing vendor will coordinate usability testing. We've hired a market research firm to line up 20 typical users for the usability study, but if you guys uncover any usability issues . . .

Vinod: I know, pass them along.

Doug: Thanks, Vinod.

25.5 Component-Level Testing

Component-level testing, also called *function testing*, focuses on a set of tests that attempt to uncover errors in WebApp functions. Each WebApp function is a software module (implemented in one of a variety of programming or scripting languages) and can be tested using black-box (and in some cases, white-box) techniques as discussed in Chapter 23.

Component-level test cases are often driven by forms-level input. Once forms data are defined, the user selects a button or other control mechanism to initiate execution. Equivalence partitioning, boundary value analysis, and path testing (Chapter 23) can be adapted for use in testing forms-based input and the functionality that is applied to it.

In addition to these test-case design methods, a technique called *forced error testing* [Ngu01] is used to derive test cases that purposely drive the WebApp component into an error condition. The purpose is to uncover errors that occur

during error handling (e.g., incorrect or nonexistent error messages, WebApp failure as a consequence of the error, erroneous output driven by erroneous input, side effects that are related to component processing).

Each component-level test case specifies all input values and the expected output to be provided by the component. The actual output produced as a consequence of the test is recorded for future reference during support and maintenance.

25.6 Navigation Testing

A user travels through a WebApp in much the same way as a visitor walks through a store or museum. There are many pathways that can be taken, many stops that can be made, many things to learn and look at, activities to initiate, and decisions to make. This navigation process is predictable in the sense that every visitor has a set of objectives when he arrives. At the same time, the navigation process can be unpredictable because the visitor, influenced by something he sees or learns, may choose a path or initiate an action that is not typical for the original objective. The job of navigation testing is (1) to ensure that the mechanisms that allow the WebApp user to travel through the WebApp are all functional and (2) to validate that each navigation semantic unit (NSU) can be achieved by the appropriate user category.

25.6.1 Testing Navigation Syntax

Quote:

"We're not lost. We're locationally challenged."

John M. Ford

The first phase of navigation testing actually begins during interface testing. Navigation mechanisms are tested to ensure that each performs its intended function. Splaine and Jaskiel [Spl01] suggest that each of the following navigation mechanisms should be tested: links and anchors of all types, redirects (when a user request a nonexistent URL), bookmarks, frames and frame sets, site maps, and the accuracy of internal search facilities.

Some of the tests noted can be performed by automated tools (e.g., link checking), while others are designed and executed manually. The intent throughout is to ensure that errors in navigation mechanics are found before the WebApp goes online.

25.6.2 Testing Navigation Semantics

In Chapter 17 a navigation semantic unit (NSU) is defined as "a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements" [Cac02]. Each NSU is defined by a set of navigation paths (called "ways of navigating") that connect navigation nodes (e.g., Web pages, content objects, or functionality). Taken as a whole, each NSU allows a user to achieve specific requirements defined by one or more use cases for a user category. Navigation testing exercises each NSU to ensure that these requirements can be achieved. You should answer the following questions as each NSU is tested:

? What questions must be asked and answered as each NSU is tested?

- Is the NSU achieved in its entirety without error?
- Is every navigation node (defined for an NSU) reachable within the context of the navigation paths defined for the NSU?

- If the NSU can be achieved using more than one navigation path, has every relevant path been tested?
- If guidance is provided by the user interface to assist in navigation, are directions correct and understandable as navigation proceeds?
- Is there a mechanism (other than the browser "back" arrow) for returning to the preceding navigation node and to the beginning of the navigation path?
- Do mechanisms for navigation within a large navigation node (i.e., a long Web page) work properly?
- If a function is to be executed at a node and the user chooses not to provide input, can the remainder of the NSU be completed?
- If a function is executed at a node and an error in function processing occurs, can the NSU be completed?
- Is there a way to discontinue the navigation before all nodes have been reached, but then return to where the navigation was discontinued and proceed from there?
- Is every node reachable from the site map? Are node names meaningful to end users?
- If a node within an NSU is reached from some external source, is it possible to process to the next node on the navigation path? Is it possible to return to the previous node on the navigation path?
- Does the user understand his location within the content architecture as the NSU is executed?

ADVICE

If NSUs have not been created as part of WebApp analysis or design, you can apply use cases for the design of navigation test cases. The same set of questions are asked and answered.

Navigation testing, like interface and usability testing, should be conducted by as many different constituencies as possible. You have responsibility for early stages of navigation testing, but later stages should be conducted by other project stakeholders, an independent testing team, and ultimately, by nontechnical users. The intent is to exercise WebApp navigation thoroughly.

INFO

Web Navigation Testing Tools

Objective: The objective of Web user navigation testing tools is to identify any broken web links or pages that are not reachable in a website.

Mechanics: The tools prompt you for the URL of a Web source and scan its markup language for links that do not return the correct type of Web resource. Some tools attempt to crawl through whole site to look for errors in deeper links.

Representative Tools:[11]

http://validator.w3.org/checklink>—Online WC3 link checker that analyzes HTML and XHTML documents for broken links.

http://www.relsoftware.com/—Download site for *Rel Link Checker Lite* a free tool for identifying broken links and orphaned files.

11 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

25.7 Configuration Testing

Configuration variability and instability are important factors that make WebApp testing a challenge. Hardware, operating system(s), browsers, storage capacity, network communication speeds, and a variety of other client-side factors are difficult to predict for each user. In addition, the configuration for a given user can change [e.g., operating system (OS) updates, new ISP and connection speeds] on a regular basis. The result can be a client-side environment that is prone to errors that are both subtle and significant. One user's impression of the WebApp and the manner in which she interacts with it can differ significantly from another user's experience, if both users are not working within the same client-side configuration.

The job of *configuration testing* is not to exercise every possible client-side configuration. Rather, it is to test a set of probable client-side and server-side configurations to ensure that the user experience will be the same on all of them and to isolate errors that may be specific to a particular configuration.

25.7.1 Server-Side Issues

On the server side, configuration test cases are designed to verify that the projected server configuration [i.e., WebApp server, database server, operating system(s), firewall software, concurrent applications] can support the WebApp without error.

As server-side configuration tests are designed, you should consider each component of the server configuration. Among the questions that need to be asked and answered during server-side configuration testing are:

? What questions must be asked and answered as server-side configuration testing is conducted?

- Is the WebApp fully compatible with the server OS?
- Are system files, directories, and related system data created correctly when the WebApp is operational?
- Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
- Has the WebApp been tested with the distributed server configuration[12] (if one exists) that has been chosen?
- Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
- Do server-side WebApp scripts execute properly?
- Have system administrator errors been examined for their effect on WebApp operations?
- If proxy servers are used, have differences in their configuration been addressed with on-site testing?

12 For example, a separate application server and database server may be used. Communication between the two machines occurs across a network connection.

25.7.2 Client-Side Issues

On the client side, configuration tests focus more heavily on WebApp compatibility with configurations that contain one or more permutations of the following components: hardware, operating systems, browser software, user interface components, plug-ins, and connectivity services (e.g., cable, DSL, WiFi). In addition to these components, other variables include networking software, the vagaries of the ISP, and applications running concurrently.

To design client-side configuration tests, you must reduce the number of configuration variables to a manageable number.[13] To accomplish this, each user category is assessed to determine the likely configurations to be encountered within the category. In addition, industry market share data may be used to predict the most likely combinations of components. The WebApp is then tested within these environments.

INFO

Web Configuration Testing Tools

Objective: The objective of Web configuration testing tools is to determine problems that may occur when a page is displayed by different web browser and operating system combinations.

Mechanics: These tools prompt you to enter the URL of a Web page and allow you to select from dozens of browser and operating systems combinations. The tools will display thumbnails of the Web page as is appears on each browser version selected.

Representative Tools:[14]

http://browsershots.org/—Browsershots provides an online service that allows you to test your website from many different browsers and operating systems.

http://testingbot.com/—*TestingBot* provides a limited free trial of an online service that allows you to test your website using many different browsers and operating systems.

25.8 Security Testing[15]

Quote:

"The Internet is a risky place to conduct business or store assets. Hackers, crackers, snoops, spoofers, . . . vandals, virus launchers, and rogue program purveyors run loose."

Dorothy and Peter Denning

WebApp security is a complex subject that must be fully understood before effective security testing can be accomplished.[16] WebApps and the client-side and server-side environments in which they are housed represent an attractive target for external hackers, disgruntled employees, dishonest competitors, and anyone else who wishes to steal sensitive information, maliciously modify content, degrade performance, disable functionality, or embarrass a person, organization, or business.

13 Conducting tests on every possible combination of configuration components is far too time-consuming.

14 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

15 Security testing is also discussed as part of security engineering in Chapter 27.

16 Books by Cross and Fisher [Cro07], Andrews and Whittaker [And06], and Trivedi [Tri03] provide useful information about the subject.

ADVICE

If the WebApp is business critical, maintains sensitive data, or is a likely target of hackers, it's a good idea to outsource security testing to a vendor who specializes in it.

Security tests are designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment. Each of these domains can be attacked, and it is the job of the security tester to uncover weaknesses that can be exploited by those with the intent to do so.

On the client side, vulnerabilities can often be traced to preexisting bugs in browsers, e-mail programs, or communication software. On the server side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client side or used to disable server operations. In addition, server-side databases can be accessed without authorization (data theft).

KEY POINT

Security tests should be designed to exercise firewalls, authentication, encryption, and authorization.

To protect against these (and many other) vulnerabilities, firewalls, authentication, encryption, and authorization techniques can be used. Security tests should be designed to probe each of these security technologies in an effort to uncover security holes.

The actual design of security tests requires in-depth knowledge of the inner workings of each security element and a comprehensive understanding of a full range of networking technologies. In many cases, security testing is outsourced to firms that specialize in these technologies.

INFO

Web Security Testing Tools

Objective: The objective of Web security testing tools is to help identify potential security problems present in a website.

Mechanics: These tools are typically downloaded and run in the development environment. They check web application for scripts that can inject harmful data that can alter the website functionality. Some tools allow you to schedule their use as probing or monitoring tools.

Representative Tools:[17]

http://www.mavitunasecurity.com/communityedition/—Download site for a tool (*Netsparker*) that checks WebApps for SQL injection vulnerabilities.

http://enyojs.com/—Download site for the free *N-Stalker* tool that performs a number of security checks on websites using the *N-Stealth* web attack signature database.

http://code.google.com/p/skipfish/—Download site for skipfish which prepares a report on security vulnerabilities found by crawling the pages in a website.

25.9 Performance Testing

Nothing is more frustrating than a WebApp that takes minutes to load content when competitive sites download similar content in seconds. Nothing is more aggravating than trying to log on to a WebApp and receiving a "server-busy"

17 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

message, with the suggestion that you try again later. Nothing is more disconcerting than a WebApp that responds instantly in some situations, and then seems to go into an infinite wait state in other situations. All of these occurrences happen on the Web every day, and all of them are performance related.

Performance testing is used to uncover performance problems that can result from a lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty or weak operating system capabilities, poorly designed WebApp functionality, and other hardware or software issues that can lead to degraded client-server performance. The intent is twofold: (1) to understand how the system responds as *loading* (i.e., number of users, number of transactions, or overall data volume), and (2) to collect metrics that will lead to design modifications to improve performance.

25.9.1 Performance Testing Objectives

Performance tests are designed to simulate real-world loading situations. As the number of simultaneous WebApp users grows, or the number of online transactions increases, or the amount of data (downloaded or uploaded) increases, performance testing will help answer the following questions:

ADVICE

Some aspects of WebApp performance, at least as it is perceived by the end user, are difficult to test. Network loading, the vagaries of network interfacing hardware, and similar issues are not easily tested at the WebApp level.

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point (in terms of users, transactions, or data loading) does performance become unacceptable?
- What system components are responsible for performance degradation?
- What is the average response time for users under a variety of loading conditions?
- Does performance degradation have an impact on system security?
- Is WebApp reliability or accuracy affected as the load on the system grows?
- What happens when loads that are greater than maximum server capacity are applied?
- Does performance degradation have an impact on company revenues?

To develop answers to these questions, two different performance tests are conducted: (1) *load testing* examines real-world loading at a variety of load levels and in a variety of combinations, and (2) *stress testing* forces loading to be increased to the breaking point to determine how much capacity the WebApp environment can handle. Each of these testing strategies is considered in the sections that follow.

25.9.2 Load Testing

ADVICE

If a WebApp uses multiple servers to provide significant capacity, load testing must be performed in a multiserver environment.

The intent of load testing is to determine how the WebApp and its server-side environment will respond to various loading conditions. As testing proceeds, permutations to the following variables define a set of test conditions:

N, number of concurrent users

T, number of online transactions per unit of time

D, data load processed by the server per transaction

In every case, these variables are defined within normal operating bounds of the system. As each test condition is run, one or more of the following measures are collected: average user response, average time to download a standardized unit of data, or average time to process a transaction. You should examine these measures to determine whether a precipitous decrease in performance can be traced to a specific combination of *N, T,* and *D*.

Load testing can also be used to assess recommended connection speeds for users of the WebApp. Overall throughput, *P,* is computed in the following manner:

$$P = N \times T \times D$$

As an example, consider a popular sports news site. At a given moment, 20,000 concurrent users submit a request (a transaction, *T*) once every 2 minutes on average. Each transaction requires the WebApp to download a new article that averages 3K bytes in length. Therefore, throughput can be calculated as:

$$P = [20{,}000 \times 0.5 \times 3\text{Kb}]/60 = 500 \text{ Kbytes/sec}$$
$$= 4 \text{ megabits per second}$$

The network connection for the server would therefore have to support this data rate and should be tested to ensure that it does.

25.9.3 Stress Testing

KEY POINT

The intent of stress testing is to better understand how a system fails as it is stressed beyond its operational limits.

Stress testing is a continuation of load testing, but in this instance the variables, *N, T,* and *D* are forced to meet and then exceed operational limits. The intent of these tests is to answer each of the following questions:

- Does the system degrade "gently," or does the server shut down as capacity is exceeded?
- Does server software generate "server not available" messages? More generally, are users aware that they cannot reach the server?
- Does the server queue resource requests and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected as capacity is exceeded?

- What values of *N, T,* and *D* force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- If the system does fail, how long will it take to come back online?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

A variation of stress testing is sometimes referred to as *spike/bounce testing* [Spl01]. In this testing regime, load is spiked to capacity, then lowered quickly to normal operating conditions, and then spiked again. By bouncing system loading, you can determine how well the server can marshal resources to meet very high demand and then release them when normal conditions reappear (so that they are ready for the next spike).

INFO

Web Performance Testing Tools

Objective: The objective of Web performance testing tools is to look for bottlenecks that can cause poor performance or simulate conditions that may cause a website to fail completely.

Mechanics: Online tools prompt you for the URL of a web resource. Some tools automatically conduct a series of simulated load tests. Some tools collect statistics on page loading and server response times as developers navigate the website.

Representative Tools:[18]

http://loadimpact.com/—*LoadImpact* is an online tool that conducts load impact testing using simulated user loads on web servers.

http://www.websitepulse.com/help/testtools.website-test.html/—*WebSitePulse* is an online tool which measures server availability and the response time of a website.

http://www.websiteoptimization.com/services/analyze/—*Web Page Analyzer* is an online tool which measures website performance and provides a list of suggested changes to improve load times.

http://developer.yahoo.com/yslow/—Yslow is an online tool that analyzes Web pages and suggests improvements based on rules for development of high-performing websites.

http://tools.pingdom.com/fpt/—*Pingdom* is an online tool which measures Web page load time bottlenecks by analyzing component elements individually.

25.10 SUMMARY

The goal of WebApp testing is to exercise each of the many dimensions of WebApp quality with the intent of finding errors or uncovering issues that may lead to quality failures. Testing focuses on content, function, structure, usability, navigability, performance, compatibility, interoperability, capacity, and security.

18 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

It incorporates reviews that occur as the WebApp is designed, and tests that are conducted once the WebApp has been implemented.

The WebApp testing strategy exercises each quality dimension by initially examining "units" of content, functionality, or navigation. Once individual units have been validated, the focus shifts to tests that exercise the WebApp as a whole. To accomplish this, many tests are derived from the user's perspective and are driven by information contained in use cases. A WebApp test plan is developed and identifies testing steps, work products (e.g., test cases), and mechanisms for the evaluation of test results. The testing process encompasses seven different types of testing.

Content testing (and reviews) focus on various categories of content. The intent is to uncover both semantic and syntactic errors that affect the accuracy of content or the manner in which it is presented to the end user. Interface testing exercises the interaction mechanisms that enable a user to communicate with the WebApp and validates aesthetic aspects of the interface. The intent is to uncover errors that result from poorly implemented interaction mechanisms or from omissions, inconsistencies, or ambiguities in interface semantics.

Navigation testing applies use cases, derived as part of the modeling activity, in the design of test cases that exercise each usage scenario against the navigation design. Navigation mechanisms are tested to ensure that any errors impeding completion of a use case are identified and corrected. Component testing exercises content and functional units within the WebApp.

Configuration testing attempts to uncover errors and/or compatibility problems that are specific to a particular client or server environment. Tests are then conducted to uncover errors associated with each possible configuration. Security testing incorporates a series of tests designed to exploit vulnerabilities in the WebApp and its environment. The intent is find security holes. Performance testing encompasses a series of tests that are designed to assess WebApp response time and reliability as demands on server-side resource capacity increase.

Problems and Points to Ponder

25.1. Are there any situations in which WebApp testing should be totally disregarded?

25.2. In your own words, discuss the objectives of testing in a WebApp context.

25.3. Compatibility is an important quality dimension. What must be tested to ensure that compatibility exists for a WebApp?

25.4. Which errors tend to be more serious—client-side errors or server-side errors? Why?

25.5. What elements of the WebApp can be "unit tested"? What types of tests must be conducted only after the WebApp elements are integrated?

25.6. Is it always necessary to develop a formal written test plan? Explain.

25.7. Is it fair to say that the overall WebApp testing strategy begins with user-visible elements and moves toward technology elements? Are there exceptions to this strategy?

25.8. Is content testing *really* testing in a conventional sense? Explain.

25.9. Describe the steps associated with database testing for a WebApp. Is database testing predominantly a client-side or server-side activity?

25.10. What is the difference between testing that is associated with interface mechanisms and testing that addresses interface semantics?

25.11. Assume that you are developing an online pharmacy (**YourCornerPharmacy.com**) that caters to senior citizens. The pharmacy provides typical functions, but also maintains a database for each customer so that it can provide drug information and warn of potential drug interactions. Discuss any special usability tests for this WebApp.

25.12. Assume that you have implemented a drug interaction checking function for **YourCornerPharmacy.com** (Problem 20.11). Discuss the types of component-level tests that would have to be conducted to ensure that this function works properly. (Note: A database would have to be used to implement this function.)

25.13. What is the difference between testing for navigation syntax and navigation semantics?

25.14. Is it possible to test every configuration that a WebApp is likely to encounter on the server side? On the client side? If it is not, how do you select a meaningful set of configuration tests?

25.15. What is the objective of security testing? Who performs this testing activity?

25.16. **YourCornerPharmacy.com** (Problem 25.11) has become wildly successful, and the number of users has increased dramatically in the first two months of operation. Draw a graph that depicts probable response time as a function of number of users for a fixed set of server-side resources. Label the graph to indicate points of interest on the "response curve."

25.17. In response to it, success **YourCornerPharmacy.com** (Problem 25.11) has implemented a special server solely to handle prescription refills. On average, 1000 concurrent users submit a refill request once every two minutes. The WebApp downloads a 500-byte block of data in response. What is the approximate required throughput for this server in megabits per second?

25.18. What is the difference between load testing and stress testing?

Further Readings and Information Sources

The literature for WebApp testing continues to evolve. Books by Andrews and Whittaker (*How to Break Web Software,* Addison-Wesley, 2006), Ash (*The Web Testing Companion,* Wiley, 2003), Nguyen and his colleagues (*Testing Applications for the Web,* 2nd ed., Wiley, 2003), Dustin and his colleagues (*Quality Web Systems,* Addison-Wesley, 2002), and Splaine and Jaskiel [Spl01] are among the most complete treatments of the subject published to date. Whitaker and his colleagues describe additional web testing practices (*How Google Tests Software,* Addison-Wesley, 2012). Mosley (*Client-Server Software Testing on the Desktop and the Web,* Prentice Hall, 1999) addresses both client-side and server-side testing issues.

Useful information of WebApp testing strategies and methods, as well as a worthwhile discussion of automated testing tools is presented by David (*Selenium 2 Testing Tools: A Beginner's Guide,* Packit Publishing, 2012) and Stottlemeyer (*Automated Web Testing Toolkit,* Wiley, 2001). Graham and her colleagues (*Experiences of Test Automation,* Addison-Wesley, 2012) and (*Software Test Automation,* Addison-Wesley, 1999) present additional material on automated tools.

Microsoft (*Performance Testing Guidance for Web Applications,* Microsoft Press, 2008) and Subraya *(Integrated Approach to Web Performance Testing,* IRM Press, 2006) present detailed treatments of performance testing for WebApps. Hope and Walther (*Web Security Testing Cookbook,* O'Reilly, 2008), Skoudis (*Counter Hack Reloaded,* Prentice Hall, 2nd ed, 2006), Andreu (*Profession Pen Testing for Wes Applications,* Wrox, 2006), Chirillo (*Hack Attacks Revealed,* 2nd ed., Wiley, 2003), Splaine (*Testing Web Security,* Wiley, 2002), and Klevinsky and his colleagues (*Hack I.T.: Security through Penetration Testing,* Addison-Wesley, 2002) provide much useful information for those who must design security tests. In addition, books that address security testing for software in general can provide important guidance for those who must test WebApps. Representative titles include: Engebretson (*Basics of Hacking and Penetration Testing,* Syngress, 2011), Basta and Halton (*Computer Security and Penetration Testing,* Thomson Delmar Learning, 2007), Wysopal and his colleagues (*The Art of Software Security Testing,* Addison-Wesley, 2006), and Gallagher and his colleagues (*Hunting Security Bugs,* Microsoft Press, 2006).

A wide variety of information sources on WebApp testing is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

26 Testing MobileApps

Key Concepts

automation..... 571
checklist..... 570
guidelines..... 568
internationalization. 578
real-time testing.. 578
strategies..... 569
stress testing 573
testing-in-the-wild .573
test matrix..... 572
tools and environments 579
usability testing .. 575

The same sense of urgency that drives WebApp projects also pervades MobileApp projects. Stakeholders are worried that they will miss a market window and press to get the MobileApp into the store. Technical activities that often occur late in the process, such as performance and security testing, are sometimes given short shrift. Usability testing that should occur during the design phase may end up being deferred until just before delivery. These can be catastrophic mistakes. To avoid this situation, you and other team members must ensure that each work product exhibits high quality. A white paper posted by Soasta [Soa11] summarizes this:

> Mobile technology is simply ramping up faster than other technologies in the past—it may be the fastest adoption curve in history. And this has important implications for your business model. You've got to be fast to market but also prepared for rapid adoption. If your application performs poorly, or fails under load, many competitors are ready to take your place—the barriers to entry are low.

MobileApp requirements and design models cannot be tested solely with executable test cases. You and your team should conduct technical reviews

Quick Look

What is it? MobileApp testing is a collection of related activities with a single goal: to uncover errors in MobileApp content, function, usability, navigability, performance, capacity, and security. To accomplish this, a testing strategy that encompasses both reviews and executable testing is applied.

Who does it? Software engineers and other project stakeholders (managers, customers, end users) all participate in MobileApp testing.

Why is it important? If end users encounter errors or difficulties within the MobileApp, they will go elsewhere for the personalized content and function they need. For this reason, you must work to find and correct as many errors as possible before the MobileApp is placed in an app store or repository.

What are the steps? The MobileApp testing process begins by focusing on user-visible aspects of the MobileApp and proceeds to testing of technology and infrastructure. Several testing steps are performed: content testing, interface testing, navigation testing, component testing, configuration testing, performance testing, and security testing.

What is the work product? A MobileApp test plan is often produced. A suite of test cases is developed for each testing step and an archive of test results is maintained for future use.

How do I ensure that I've done it right? Although you can never be sure that you've performed every test that is needed, you can be certain that testing has uncovered errors (and that those errors have been corrected). In addition, if you've established a test plan, you can check to ensure that all planned tests have been conducted.

(Chapter 20) and test usability (Chapter 15), as well as performance. Configuration testing is especially important as a mechanism for verifying the MobileApp's ability to take context into account. The intent is to uncover and correct errors before the MobileApp is released to the end-user community.

There are several important questions to ask when creating a MobileApp testing strategy [Sch09].

- Do you have to build a fully functional prototype before you test with users?
- Should you test with the user's device or provide a device for testing?
- What devices and user groups should you include in testing?
- What are the benefits/drawbacks of lab testing versus remote testing?

We address each of these questions throughout this chapter.

26.1 Testing Guidelines

MobileApps that execute entirely on a mobile device can be tested using traditional software testing methods (Chapter 23) or using emulators running on personal computers. On the other hand, thin-client MobileApps that make use of server-based resources are particularly challenging to test. In addition to many of the testing challenges presented by WebApps (Chapter 25), the testing of thin-client MobileApps must also consider transmission of data through Internet gateways and telephone networks [Was10]. Users expect MobileApps to be context aware and deliver personalized user experiences based on the physical location of a device in relation to available network features. But testing MobileApps in a dynamic ad hoc network environment using every possible device and network configuration is difficult, if not impossible.

ADVICE

Involve users as early as possible in the testing cycle—early feedback helps in design correction.

To understand the objectives of MobileApp testing, you should consider the many unique challenges facing app designers. MobileApps are expected to deliver much of the complex functionality and reliability found in PC-based applications, but they are resident on mobile platforms with relatively limited resources. The following guidelines provide a basis for mobile application testing [Kea07]:

- *Understand the network and device landscape before testing to identify bottlenecks.* Testing across borders is discussed in Section 26.4.
- *Conduct tests in uncontrolled real-world test conditions* (field-based testing), especially for a multitier mobile application. Testing in the production environment is discussed in Section 26.2.5.
- *Select the right automation test tool.* Ideally, the tool should support all desired platforms, allow testing for various screen types, resolutions, and input mechanisms—such as touchpad and keypad, and implement

connectivity to the external system to carry out end-to-end testing. MobileApp testing tools are discussed in greater detail in Section 26.6.

- *Use the Weighted Device Platform Matrix method to identify the most critical hardware/platform combination to test.* This method is very useful especially when hardware/platform combinations are numerous and time to test is low. Details of applying this method are described in Section 26.2.3.
- *Check the end-to-end functional flow in all possible platforms at least once.* When Web services are involved, it is difficult to trace the actual network path required to deliver a MobileApp function without performance tools. Tool use is discussed in Section 26.6.
- *Conduct performance testing, GUI testing, and compatibility testing using actual devices.* Even though these tests can be done using emulators, testing with actual devices is recommended. User interaction testing is discussed in Section 26.3 and performance issues are discussed in Section 26.2.
- *Measure performance only in realistic conditions of wireless traffic and user load.* Real-time testing issues for MobileApps are discussed in Section 26.5.

26.2 The Testing Strategies

Quote:

"I want to be buried with a mobile phone, just in case I'm not dead"

Amanda Holden

Technology alone is not sufficient to guarantee commercial success of a MobileApp. Users abandon MobileApps quickly if they do not work well or fail to meet expectations. It is important to recall that testing has two important goals: (1) to create test cases that uncover defects early in the development cycle and (2) to verify the presence of important quality attributes. The quality attributes for MobileApps are based on those set forth in ISO 9126 [Spr04] and encompass functionality, reliability, usability, efficiency, maintainability, and portability (Chapter 19).

Developing a MobileApp testing strategy requires an understanding of both software testing and the challenges that make mobile devices and their network infrastructure unique [Kho12a]. In addition to a thorough knowledge of conventional software testing approaches (Chapters 22 and 23), a MobileApp tester should have a good understanding of telecommunications principles and an awareness of the differences and capabilities of mobile operating systems platforms. This basic knowledge must be complemented with a thorough understanding of the different types of mobile testing (e.g., MobileApp testing, mobile handset testing, mobile website testing), the use of simulators, test automation tools, and remote data access services (RDA). Each of these topics is discussed later in this chapter.

INFO

MobileApp Testing—Checklist

Nari Kannnan, CEO of the mobile applications consulting company appsparq, Inc. posted the following recommendations for testing MobileApps [Kan11]:

- **Conceptual testing**—Getting input from prospective users of the MobileApp prior to beginning development helps ensure that no unnecessary features are included and no essential features have been missed. (Requirements gathering and validation techniques are discussed in Chapter 9.)
- **Unit and system testing**—Implementing regular unit and system testing accommodates any software application that contains multiple components and interacts with networks (Chapters 22 and 26).
- **User experience testing**—Involving real users early in the development process ensures that the experience provided lives up to the usability and accessibility expectations of its stakeholders. MobileApp usability testing is discussed in Section 26.3.
- **Stability testing**—Ensuring that the MobileApp does not crash owing to incompatibilities with network or Web services. Testing MobileApps in production environments is discussed in Section 23.2.5.
- **Connectivity testing**—Testing access to all essential components and Web resources is essential to ensuring the MobileApp is making appropriate use of context. Testing across borders is discussed in Section 26.6.
- **Performance testing**—Testing the ability of the MobileApp to meet its nonfunction requirements (download times, processor speed, storage capacity, etc.). Performance testing is discussed in Sections 26.2.4 and 26.5.
- **Device compatibility testing**—Verifying that the MobileApp runs correctly on all targeted devices. The task is discussed in Section 26.2.3.
- **Security testing**—Ensuring the MobileApp meets the security requirements set by its stakeholders. Security assurance testing is discussed in Chapter 27.
- **Certification testing**—Checking that the MobileApp meets the standards established by the app stores that will distribute it.

26.2.1 Are Conventional Approaches Applicable?

KEY POINT

Those who must test MobileApps are well served by adopting the overall strategy for testing WebApps

A comprehensive MobileApp testing program includes the generic spiral approach discussed in Chapter 22, but will also include adaptations discussed for client-server architectures, real-time computing, graphical user interfaces, WebApps, and object-oriented systems (Chapters 23–25). MobileApp testing also has unique challenges that should be addressed to ensure that an app meets both its functional and nonfunctional requirements.[1]

Vinson [Vin11] suggests that MobileApp testers adapt the strategy used for testing WebApps (Chapter 25). Content must be tested to be sure that it was chosen with the limitations of mobile devices and ad hoc networks in mind. Compatibility testing and deployment testing are more challenging in the mobile world, owing to the large variety in device characteristics and user environments. Performance testing needs to determine whether the limited storage, processing, connectivity, and power available on a mobile device may negatively impact features or functionality. MobileApp performance testing is often conducted at a level of detail

1 An overview of the unique issues in testing MobileApps can be found at: **http://www.utest.com/landing-interior/crowdsource-your-mobile-app-testing**

that is seen only in the development of real-time systems. Security testing has to take the loss of a physical device into account as well as the fact that MobileApps often run directly on the mobile device hardware exposing personal data to theft. MobileApps are often designed to be used by people with less technical knowledge than the typical Web user precipitating the need for more extensive testing of the user experience (Section 26.3). Agile development process models (Chapter 5) and/or test-driven development models (Chapter 24) can be used.

26.2.2 The Need for Automation

Quote:

"Today there are hundreds of millions of mobile devices, but you do have to know a bit about what each device is capable of doing in order to approach it as a developer."

John Fowler

A MobileApp tester often encounters many configuration variants (devices, operating systems, and mobile networks) that may need to be included to ensure that a MobileApp is both functional and context aware. Because it is important to test a MobileApp efficiently and completely, automated testing can be useful for configuration testing (Section 25.7) or regression testing (Section 22.3). It should be noted, however, that it may not be possible to automate all parts of MobileApp testing (e.g., user interactions found in a handheld video game).

Automated testing tools can improve team morale when testers would otherwise be forced to process a large number of repetitive test cases mechanically. The availability of automated test tools can encourage earlier and more frequent testing, allowing the early discovery of MobileApp defects. An agile development process (Chapter 5) mandates the use of daily build cycles that require regression testing to ensure that changes have not produced unintended side effects.

Mobile Labs, Inc. [Mob11] has proposed an approach to automating MobileApp testing that encompasses the following elements:

? What are the most important elements of automated mobile testing?

Feasibility analysis. Determines which tests and test cases have the greatest return on investment (ROI) if automated. The focus should be on automating repeatable and frequently used test cases. The goal is to try to automate 50 to 60 percent of manual test cases.

Proof of concept. Validates the value of test automation. To accomplish this, a limited number of manual test scripts are automated to determine the ROI of the effort. The test team must determine scalability to the other scripts and the degree of reuse in subsequent testing cycles.

Best practice test framework. Provides a methodology specific to mobile applications that serves as the foundation for the testing process. Frameworks define the rules for implementation and testing of the MobileApp and are developed for each mobile platform and tailored to the organization's application suite.

Customized testing tools. Customizes testing tools to each mobile platform (and the application being tested) by leveraging advanced scripting techniques.

Testing under real-world conditions. Confirms how the application will run on an actual device outside the testing laboratory. Testing on actual devices

instead of emulators reduces the incidence of false defect reporting and ensures that user-level errors will be more likely to be uncovered (Section 26.2.5).

Rapid defect resolution. Speeds implementation through automatic submission of defect information and generation of discrepancy reports allowing developers to reduce the time required to resolve defects.

Reuse of test scripts. Provides cost savings by eliminating the need to start test case creation from scratch when enhancements are made. It is important for the test tool architecture to allow the separation of function interfaces and test logic. This allows interfaces to be wrapped in reusable functions as tools are adapted for new platforms and devices.

26.2.3 Building a Test Matrix

MobileApps are often developed for multiple devices and designed to be used in many different contexts and locations. A *weighted device platform matrix* (WDPM) helps ensure that test coverage includes each combination of mobile device and context variables.[2] The WDPM can also be used to help prioritize the device/context combinations so that the most important are tested first.

The steps to build the WDPM (Figure 26.1) for several devices and operating systems are: (1) list the important operating system variants as the matrix column labels, (2) list the targeted devices as the matrix row labels, (3) assign a ranking (e.g., 0 to 10) to indicate the relative importance of each operating system and each device, and (4) compute the product of each pair of rankings and enter each product as the cell entry in the matrix (use NA for combinations that are not available).

Testing effort should be adjusted so that the device/platform combinations with the highest ratings receive the most attention for each context variable under consideration. In Figure 26.1, **Device4** and **OS3** have the highest rating and would receive high-priority attention during testing.

FIGURE 26.1

Weighted device platform matrix

		OS1	OS2	OS3
	Ranking	3	4	7
Device 1	7	NA	28	49
Device 2	3	9	NA	NA
Device 3	4	14	NA	NA
Device 4	9	NA	36	63

2 *Context variables* are variables that are associated with either the current connection or the current transaction that the MobileApp will use to direct its visible-user behavior.

26.2.4 Stress Testing

KEY POINT

Graceful degradation is an important property of fault tolerant systems. A system that degrades gracefully will attempt to get to a known safe state in the advent of an error before shutting down if it cannot repair the damage and allow execution to continue.

Stress testing for mobile apps attempts to find errors that will occur under extreme operating conditions. In addition, it provides a mechanism for determining whether the MobileApp will degrade gracefully without compromising security. Among the many actions that might create extreme conditions are: (1) running several mobile apps on the same device, (2) infecting system software with viruses or malware, (3) attempting to take over a device and use it to spread spam, (4) forcing the mobile app to process inordinately large numbers of transactions, and (5) storing inordinately large quantities of data on the device. As these conditions are encountered, the MobileApp is checked to ensure that resource intensive services (e.g., streaming media) are handled properly.

Effective stress testing [Soa11] should be performed "in the wild," where different devices and different operating environments are common. Conditions that exercise two to three times rated capacity should be tested. Tests should reflect actual user behavior in real-life contexts and should reflect users who change location locally, as well as users in other countries with different network configuration and standards.

In the sections that follow, we consider some of these guidelines in greater detail.

26.2.5 Testing in a Production Environment

Many MobileApp developers advocate *testing-in-the-wild*, or testing in the users' native environments with the production release versions of the MobileApp resources. Testing-in-the-wild is designed to be agile and respond to changes as the MobileApp evolves [Ute12].

Quote:

" The important thing about mobile is that everybody has a computer in their pocket."

Ben Horowitz

Some of the characteristics of in-the-wild testing include adverse and unpredictable environments, outdated browsers and plug-ins, unique hardware, and imperfect connectivity (both Wi-Fi and mobile carrier). In order to mirror real-world conditions, the demographic characteristics of testers should match those of targeted users, as well as those of their devices. In addition, you should include use cases involving small numbers of users, less-popular browsers, as well as a diverse set of mobile devices. Testing-in-the-wild is always somewhat unpredictable, and test plans must be adapted as testing progresses. For further information, Rooksby and his colleagues have identified themes that are present in successful strategies for testing-in-the-wild [Roo09].

Creating test environments in-house is an expensive and error-prone process. Cloud-based testing can offer a standardized infrastructure and preconfigured software images, freeing the MobileApp team from the need to worry about finding servers or purchasing their own licenses for software and testing tools. Cloud service providers give testers access to scalable and ready to user virtual laboratories with a library of operating systems, test and execution management tools, and storage necessary for creating a test environment that closely mirrors the real world. [Myl11].

Cloud-based testing is not without potential problems: lack of standards, potential security issues, data location and integrity issues, incomplete infrastructure support, improper usages of services, and performance issues are only some of the common challenges that face development teams that use the cloud approach.

SafeHome

MobileApp Testing in the Production Environment

The scene: Doug Miller's office.

The players: Doug Miller (manager of the *SafeHome* software engineering group) and Vinod Raman (a member of the product software engineering team).

The conversation:

Doug: What do you think of the e-commerce portion of our **SafeHomeAssured** MobileApp V0.0?

Vinod: The outsourcing vendor has done a good job of adapting the WebApp **SafeHomeAssured.com** to the mobile environment. Sharon [development manager for the vendor] tells me they're testing the prototype as we speak.

Doug: I heard they were doing testing for the e-commerce site using device emulators. I think we should do a little testing on actual devices.

Vinod (grimacing): I thought we were going to hire a third-party testing company to validate the MobileApp. We're still killing ourselves trying to get the product software out the door.

Doug: We're going to hire a testing vendor for performance, security testing, and configuration testing. Our outsourcing vendor is already doing some testing. I just thought another point of view would be helpful, and besides, we'd like to keep costs in line, so . . .

Vinod (sighs): What are you looking for?

Doug: I want to be sure that the user experience is solid.

Vinod: I suppose we can start with the use cases for each of the major interface functions.

Doug: Good. But follow the logic paths from their beginning to their conclusion. Take a look at the weighted device platform matrix. I'd like you to check its performance on the top six most important devices, and while you're there, check out the content that appears at each navigation node. Make sure it takes the device characteristics into account as each screen display is rendered.

Vinod: Of course . . . and the functional elements as well. Who's testing usability?

Doug: Oh . . . the testing vendor will coordinate usability testing. We've hired a market research firm to line up 20 typical users for the usability study, but if you guys uncover any usability issues . . .

Vinod: I know, pass them along.

Doug (smiling): Thanks, Vinod.

26.3 Considering the Spectrum of User Interaction

? What characteristics of MobileApp usability become the focus of testing and what specific objectives are addressed?

In a crowded marketplace in which products provide the same functionality, users will choose the MobileApp that is easiest to use. The user interface and its interaction mechanisms are visible to the MobileApp users. It is important to test the quality of the user experience provided by the MobileApp to ensure that it meets the expectations of its users.

Many of the procedures for assessing the usability of software user interfaces discussed in Chapter 15 can be used to assess MobileApps. Similarly many of the strategies used to assess the quality of WebApps (Chapter 25) may be used to test

the user interface portion of the MobileApp. There is more to building a good MobileApp user interface than simply shrinking the size of a user interface from an existing personal computer application.

INFO

MobileApp Usability Testing Components

A number of recommendations for key components MobileApp usability testing have been presented in the MobileApp Testing Blog.[3]

- **Functionality**—Ensure that core functionality is supported by the user stories and takes stakeholder goals and expectations into account.
- **Information architecture**.—Make sure that content and links have been structured and presented in a logical manner, taking chunking[4] and perspectives into account.
- **Content**—Use text, video, images, and multimedia only when it supports the user task in a mobile context, give user control over whether to start media or not, make sure the content is presented in mobile device format.
- **Design**—Design for quick visual scanning of screen, consider both portrait and landscape display orientations, rethink the screen layout, do not just shrink it.
- **User input**—Make it easy for users to enter data, offer auto completion and spell checking, display default values, offer alternate input mechanisms based on individual device capabilities.
- **Mobile context**—Account for changes in context (time of day, location, networks) and use device features and capabilities to anticipate and support the user's context of use.
- **Usability**—Ensure that interaction devices (touch screens, keyboards, audio) and widgets (buttons, links, scrollbar) work well together on all targeted devices, follow conventions, and reduce the learning curve.
- **Trustworthiness**—Be sensitive to privacy and security, do not collect personal information without explicit user permission, allow user to control how personal info is shared and state your business practices.
- **Feedback**—Get important information to your users, minimize number of alerts, make alerts brief and informative, provide confirmation of user actions without disrupting user's workflow.
- **Help**—Make it easy for users to access help and support options, offer contextual help.

26.3.1 Gesture Testing

Touch screens are very popular on mobile devices and, as a consequence, developers have added multitouch gestures (e.g., swiping, zooming, scrolling, selection) as a means augmenting the user interaction possibilities without losing screen real estate. Unfortunately, gesture-intensive interfaces present a number of testing challenges.

It is difficult to use automated tools to test touch or gesture interface actions. Gestures are hard to log accurately for replay. The location of screen objects is affected by screen size and resolution, as well as previous user actions. Paper prototypes, sometimes developed as part of design, cannot be used to adequately

3 The blog can be found at **http://www.mobileapptesting.com/10-key-components-of-successful-mobile-app-usability/2012/09/**

4 *Chunking* is the practice of breaking up hypermedia documents into smaller groupings of related information to allow faster assimilation by the reader.

test gestures. Instead, testers need to create test framework programs that make calls to functions that simulate gesture events. All of this is expensive and time-consuming.

Accessibility testing for visually impaired users is challenging because gesture interfaces typically do not provide either tactile or auditory feedback. Usability and accessibility testing for gestures become very important for ubiquitous devices like smartphones. It may be important to test the operation of the device when gesture operations are not available.

Ideally user stories or use cases are written in sufficient detail to allow their use as the basis for test scripts. It is important to recruit representative users and include all targeted devices to take screen differences into account when testing gestures with a MobileApp. Finally, testers should ensure that the gestures conform to the standards and contexts set for the mobile device or platform.

26.3.2 Voice Input and Recognition

Smart mobile devices now make use of voice input to allow for contemporaneous hands-busy, eyes-busy operation of the device. Voice input may take several forms with different levels of programming complexity required to process each. Voice-mail input occurs when a message is simply recorded for playback later. Discrete word recognition can be used to allow users to verbally select items from a menu with a small number of choices. Continuous speech recognition reflects attempts to translate dictated speech into meaningful text strings. Each type of voice input has its own testing challenges.

ADVICE

Testing of voice input should take environmental conditions and individual voice variation into account.

According to Shneiderman [Shn09] all forms of speech input and processing are hindered by interference from noisy environments. Using voice commands to control a device impresses a greater cognitive load on the user, as compared to pointing to a screen object or pressing a key. The user must think of the correct word or words to get the MobileApp to perform the desired action. When pointing at an object, however, the user merely needs to recognize the appropriate screen object and select it. However, the breadth and accuracy of speech recognition systems are evolving rapidly, and it is likely that voice recognition will become the dominant form of communication in many MobileApps.

Testing the quality and reliability of voice input and recognition presents technical challenges for even the best testing organizations. Erroneous voice input (due to user error, misspoken words or phrases, or environmental interference) must be tested to ensure that bad input does not cause the MobileApp or the device to fail. Because each user/device combination is different, a broad population of users and environments should be involved to ensure an acceptable error rate. Finally, it is important to log errors so that developers can improve the ability of the MobileApp to process speech input.

26.3.3 Virtual Key Board Input

Because a virtual keyboard may obscure part of the display screen when activated, it is important to test the MobileApp to ensure that important screen information is not hidden from the user while typing. If the screen information must be hidden it is important to test the ability of the MobileApp to allow page flipping by the user without losing typed information [Sch09].

Virtual keyboards are typically smaller than personal computer keyboards, and therefore, it is difficult to type with 10 fingers. Because the keys themselves are smaller and harder to hit and provide no tactile feedback, the MobileApp must be tested to ensure that it allows easy error correction and can manage mistyped words without crashing.

Predictive technologies (i.e., auto completion of partially typed words) are often used with virtual keyboards to help expedite user input. It is important to test the correctness of the word completions for the natural language chosen by the user, if the MobileApp is designed for a global market. It is also important to test the usability of any mechanism that allows the user to override a suggested completion.

Virtual keyboard testing is often conducted in the usability laboratory, but some should be conducted in the wild. If virtual keyboard tests uncover significant problems, the only alternative may be to ensure that the MobileApp can accept input from devices other than a virtual keyboard (e.g., a physical keyboard or voice input).

26.3.4 Alerts and Extraordinary Conditions

As a MobileApp runs in a real-time environment, there are factors that may impact its behavior. For example, a Wi-Fi signal may be lost, or an incoming text message, phone call, or calendar alert may be received while the user is working with the MobileApp.

These factors can disrupt the MobileApp user's workflow, yet most users opt to allow alerts and other interruptions as they work. Your MobileApp test environment must be able to simulate these alerts and conditions. In addition, you should test the MobileApp's ability to handle alerts and conditions in production environment on actual devices (Section 26.5).

Part of MobileApp testing should focus on the usability issues relating to alerts and pop-up messages. Testing should examine the clarity and context of alerts, the appropriateness of their location on the device display screen, and when foreign languages are involved, verification that the translation from one language to another is correct.

Many alerts and conditions may be triggered differently on various mobile devices or by network or context changes. While many of the exception-handling processes can be simulated with a software test harness, you cannot rely solely on testing in the development environment. This again emphasizes the importance of testing the MobileApp in the wild on actual devices.

26.4 Test Across Borders

Quote:

"The world is being re-shaped by the convergence of social, mobile, cloud, big data, community and other powerful forces. The combination of these technologies unlocks an incredible opportunity to connect everything together in a new way and is dramatically transforming the way we live and work.

Marc Benioff

Internationalization is the process of creating a software product so that it can be used in several countries and with various languages without requiring any engineering changes. *Localization* is the process of adapting a software application for use in targeted global regions by adding locale-specific requirements and translating text elements to appropriate languages. Localization effort may involve taking each country's currency, culture, taxes, and standards (both technical and legal) into account in addition to differences in languages [Sla12]. Launching a MobileApp in many parts of the world without testing it there would be very foolish.

Because it can be very costly to build an in-house testing facility in each country for which localization is planned, outsourcing testing to local vendors in each country is often more cost effective [Reu12]. However, using an outsourcing approach risks a degradation of communication between the MobileApp development team and those who are performing localization tests.

Crowdsourcing has become popular in many online communities.[5] Revenui [Reu12] suggests that crowdsourcing could be used to engage localization testers dispersed around the globe outside of the development environment. To accomplish this, it is important to find a community that prides itself on its reputation and has a track record of successes. An easy-to-use real-time platform allows community members to communicate with the project decision makers. To protect intellectual property, only trustworthy community members who are willing to sign nondisclosure agreements are allowed to participate.

26.5 Real-Time Testing Issues

Actual mobile devices have inherent limitations precipitated by the combination of hardware and firmware delivered in the device. If the range of potential device platforms is large, it is expensive and time-consuming to perform MobileApp testing.

Mobile devices are not designed with testing in mind. Limited processing power and storage capacity may not allow loading of the diagnostic software needed to record the test-case performance. Emulated devices are often easier to manage and allow easier acquisition of test data.

Each mobile network (there are hundreds, worldwide) uses its own unique infrastructure to support the mobile Web. The Web proxies implemented by mobile Web operators dictate how, when, and whether you can connect to specific Web resources using their network. This can restrict the flow of information that

[5] *Crowdsourcing* is a distributed problem solving model where community members work on solutions to problems posted to the group.

travels between your server and the test client. Some proxies may strip vital information from http headers your application needs to provide functionality or device adaptation. Network signal strength can be an issue. Emulators often cannot emulate the effects and timing of network services, and you may not see problems that users will have when the MobileApp runs on an actual device.

A remote mobile device is a useful piece of test equipment that can be used to overcome some of the limitations of using emulators. A *remote mobile device* is a physical mobile handset that is mounted in a box with a remote control unit and remote antenna. The remote control unit is connected to the device screen and keyboard control circuits. When connected to the Internet, this solution allows a user on a local PC or Web client to press buttons and collect data on what is happening on the remote device. Some remote devices have the ability to record test cases for subsequent replay to assist the process of automating regression testing.

Last, it is important to monitor power consumption specifically associated with the use of the MobileApp on a mobile device. Transmitting information from mobile devices consumes more power than monitoring a network for a signal. Processing streaming media consumes more power than loading a Web page or sending a text message. Assessing power consumption accurately must be done in real time on the actual device and in the wild.

26.6 Testing Tools and Environments

In Section 26.3.2 we discussed the reasons for automating some aspects of MobileApp testing in order to reduce testing time and improve the quality and coverage of the testing process. Likewise, we discussed the importance of testing in the production environment in Section 26.2.5. There are times, however, when manual testing is required, but even in those cases, tools can be used to monitor the behavior of MobileApps and users on devices across networks.

Khode [Kho12b] suggests several criteria to use when selecting mobile test automation tools. These criteria might also be applicable to mobile testing tools in general.

? What criteria should I use to select automation tools for mobile testing?

- *Object identification*—The tool can recognize device objects using a variety of methods (e.g., object ID, image processing, text recognition, HTML5/DOM objects).
- *Security*—The tool should not require the use of an unprotected device connected to the public Internet.
- *Devices*—The tool makes use of actual user devices without requiring the use of special developer modes.
- *Functionality*—All device functionality is supported including multitouch gestures, virtual keyboard input, incoming calls and text messaging, alert processing, and others.

- *Emulators and plug-ins*—The same test can be executed on different devices and different mobile operating systems using the existing testing environment.
- *Connectivity*—Simultaneous connection of multiple devices using USB, Wi-Fi, private cloud, and phone carrier to test connection stability and recovery.

SOFTWARE TOOLS

Selected Tools for MobileApp Testing

Here is a list of several tools that might be useful in MobileApp testing. This is a very volatile field. This list of representative tools was recently recommended by Brown [Bro11] and Vinson [Vin11].[6]

Mobile Web page tools try to determine the degree to which a Web page is mobile device friendly. User enters a Web URL and the tool provides a list of defects.

Representative Tool

WC3mobileOKChecker **http://validator.w3.org/mobile/**)

Mobile browser emulators show the appearance of a Web page on mobile browsers. User enters a Web URL and tool shows how it would appear on a mobile browser.

Representative Tools

Mobile Phone Emulator **http://www.mobilephoneemulator.com/**)

iPhoney **http://www.marketcircle.com/iphoney/**)

Device emulators are virtual devices that typically run on a personal computer and allow you to develop and test MobileApps without access to physical devices.

Representative Tools

Android Emulators (**http://developer.android.com/sdk/index.html**)

iPad Peek (**http://ipadpeek.com/**)

Adobe Edge Inspect **http://html.adobe.com/edge/inspect/**)

Blackberry Simulators (**http://us.blackberry.com/sites/developers/resources/simulators.html**)

Automated tools record interactions on IOS or Android and allows their play back as test scripts. Typically these run on a personal computer with a device emulator.

Representative Tools

MonkeyTalk (**http://www.gorillalogic.com/testing-tools/monkeytalk**)

Eggplant Mobile (**http://www.testplant.com/**)

Device Anywhere (**http://www.keynotedeviceanywhere.com/**

Network monitoring tools add, modify, and filter HTTP request headers sent to Web servers. Installs as a browser plug-in.

Representative Tool

Modify headers (**https://addons.mozilla.org/en-us/firefox/addon/modify-headers/**)

Mobile analytics testing collects data to allow analysis of how users interact with the MobileApp which is important to assess ROI (return on investment). Typically requires a Web or cloud service to assist in data collection and storage.

Representative Tools

Flurry (**http://www.flurry.com/flurry-analytics.html**)

Google Mobile Analytics (**http://www.google.com/analytics/mobile/**)

Distimo Monitor (**http://monitor.distimo.com/**)

[6] Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In addition, tool names are registered trademarks of the companies noted.

26.7 Summary

The goal of MobileApp testing is to exercise each of the many dimensions of MobileApp quality with the intent of finding errors or uncovering issues that may lead to quality failures. Testing focuses on quality elements such as content, function, structure, usability, use of context, navigability, performance, power management, compatibility, interoperability, capacity, and security. It incorporates reviews and usability assessments that occur as the MobileApp is designed, and tests that are conducted once the MobileApp has been implemented and deployed on an actual device.

The MobileApp testing strategy exercises each quality dimension by initially examining "units" of content, functionality, or navigation. Once individual units have been validated, the focus shifts to tests that exercise the MobileApp as a whole. To accomplish this, many tests are derived from the user's perspective and are driven by information contained in use cases. A MobileApp test plan is developed and identifies testing steps, work products (e.g., test cases), and mechanisms for the evaluation of test results. The testing process encompasses several different types of testing.

Content testing (and reviews) focus on various categories of content. The intent is to examine errors that affect the presentation of the content to the end user. The content needs to be examined for performance issues imposed by the mobile device constraints. Interface testing exercises the interaction mechanisms that define the user experience provided by the MobileApp. The intent is to uncover errors that result when the MobileApp does not take device, user, or location context into account.

Navigation testing is based on use cases, derived as part of the modeling activity. The test cases are designed to exercise each usage scenario against the navigation design within architectural framework used to deploy the MobileApp. Component testing exercises content and functional units within the MobileApp.

Configuration testing attempts to uncover errors and/or compatibility problems that are specific to a particular device or network environment. Tests are then conducted to uncover errors associated with each possible configuration. This is complicated by the large number of mobile devices and network service providers. Security testing incorporates a series of tests designed to exploit vulnerabilities in the MobileApp or its environment. The intent is to find security holes in either the device operating environment or the Web services being accessed. Performance testing encompasses a series of tests that are designed to assess MobileApp response time and reliability as demands on server-side resource capacity increase. Finally, MobileApp testing should address performance issues such as power usage, processing speed, memory limitations, ability to recover from failures, and connectivity issues.

Problems and Points to Ponder

26.1. Are there any situations in which MobileApp testing on actual devices can be disregarded?

26.2. In your own words, discuss the objectives for testing a MobileApp.

26.3. Compatibility is an important quality dimension. What must be tested to ensure that compatibility exists for a MobileApp?

26.4. Locate a free MobileApp testing tool. Critique the performance of the tool relative to the MobileApp with which you are familiar.

26.5. What elements of the MobileApp can be "unit tested"? What types of tests must be conducted only after the MobileApp elements are integrated?

26.6. Is it always necessary to develop a formal written-test plan? Explain.

26.7. Is it fair to say that the overall MobileApp-testing strategy begins with user-visible elements and moves toward technology elements? Are there exceptions to this strategy?

26.8. Is certification testing *really* testing in a conventional sense? Explain.

26.9. Describe the steps associated with user experience testing for a MobileApp.

26.10. Assume that you are developing a MobileApp to access an online pharmacy that caters to senior citizens. The pharmacy provides typical functions, but also maintains a database for each customer so that it can provide drug information and warn of potential drug interactions. Discuss any special usability tests for this MobileApp.

26.11. Assume that you have implemented a Web service that provides a drug interaction–checking function for **YourCornerPharmacy.com** (see Problem 26.10). Discuss the types of component-level tests that would have to be conducted on the mobile device to ensure that the MobileApp accesses this function properly.

26.12. How can a MobileApp's ability to take context into account be tested?

26.13. Is it possible to test every configuration that a MobileApp is likely to encounter in the production environment? If it is not, how do you select a meaningful set of configuration tests?

26.14. Describe a security test that might need to be conducted for the **YourCornerPharmacy** (Problem 26.10) MobileApp. Who should perform this test?

26.15. What is the difference between stress-testing a MobileApp and stress-testing a WebApp?

Further Readings and Information Sources

There are many books that describe mobile computing and these often contain discussions of MobileApp testing. Kumar and Xie (*Handbook of Mobile Systems Applications and Services,* Auerbach Publications, 2012) have edited a book that covers mobile services and the role of service-oriented architectures in mobile computing. Books on pervasive computing by Chalmers (*Sensing and Systems in Pervasive Computing: Engineering Context Aware Systems* Springer, 2011)), Adelstein (*Fundamentals of Mobile and Pervasive Computing*, McGraw-Hill, 2004) and Hansmann (*Pervasive Computing: The Mobile World*, 2nd ed., Springer, 2003) define the principles of context in mobile computing. The book by Nguyen, et al. (*Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems,* 2nd ed., Wiley, 2003) discusses testing of MobileApps with an emphasis on testing accessibility, reliability, and security.

Interface design books are plentiful. Many of them contain information on testing MobileApp usability. The book by Ben Shneiderman and colleagues (*Designing the User Experience*, 5th ed., Addison-Wesley, 2009) is a classic work on usability. Other good general references are books by Quesenberry and Szuc (*Global UX: Design and Research in a Connected World,* Morgan Kaufmann, 2011) and Schumacher (*Handbook of Global User Research*, Morgan Kaufmann, 2009). Nielsen (*Mobile Usability*, New Riders, 2012) offers advice on how to design usable interfaces that take mobile-device screen size into account. Colborne (*Simple and Usable Web, Mobile, and Interaction Design*, New Riders, 2010) describes the process of simplifying user interaction. Ginsburg (*Designing the iPhone User Experience: A User-Centered Approach to Sketching and Prototyping iPhone Apps*, Addison-Wesley, 2010) discusses the importance of taking a user-centric approach to assessing the user experience.

Meier (The *Microsoft Application Architecture Guide*, 2nd ed. Microsoft Press, 2009) has edited a book that contains useful information on mobile application testing. A book by Graham (*Experiences of Test Automation: Case Studies of Software Test Automation,* Addison-Wesley, 2012) provides good background on testing automation. Lee (*Test-Driven iOS Development*, Addison-Wesley, 2012) discusses the MobileApp testing process in the context of test-driven design.

A wide variety of information sources on WebApp testing is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

27 Security Engineering

Key Concepts

asset 585
assurance case ... 592
attack 585
  pattern 589
  surface 596
exposure 585
security
  assurance 592
  case 592
  engineering ... 588
  model 590
  requirements .. 585
threat
  analysis 585
  modeling 594
trust verification .. 591

Devanbu and Stubblebine [Dev00] make the following comment as an introduction to their roadmap for security engineering:

> Is there such a thing anymore as a software system that doesn't need to be secure? Almost every software controlled system faces threats from potential adversaries, from Internet-aware client applications running on PCs, to complex telecommunications and power systems accessible over the Internet, to commodity software with copy protection mechanisms. Software engineers must be cognizant of these threats and engineer systems with credible defenses, while still delivering value to customers.

The threats that the authors discussed more than a decade ago have multiplied with the explosive growth of the Web, the ubiquitous presence of mobile applications, and the widespread use of the cloud. Each of these technologies has raised new concerns about user privacy and the potential loss or theft of personal information. Security is not just a concern to the people

Quick Look

What is it? Security engineers build systems that have the ability to protect their assets from attack. Using threat analysis, you can determine the controls required to reduce the exposure that results when attacks exploit system vulnerabilities. Security is an essential prerequisite for software quality factors such as integrity, availability, reliability, and safety.

Who does it? Software engineers working with customers and any other stakeholders who rely on the system results or services.

Why is it important? Each emerging technology brings new concerns about user privacy and the potential loss or theft of valuable information. Security is not just a concern to the people developing software for the military, government, or health agencies. Today, security must be the concern of every software engineer who has client resources to protect.

What are the steps? First, system assets are identified and the exposure to loss due to security breach is determined. The system architecture is then modeled at the component level. Next, a security requirements specification and a risk mitigation plan are created. As the system is constructed, security assurance is preformed and continues throughout the software process.

What is the work product? The primary work products are a security specification (which may be part of a requirements model) and a documented security case that is part of the system quality assurance documents. To develop these work products, a threat model and security risk assessment and risk mitigation plan are also created.

How do I ensure I have done it right? Use the evidence resulting from security reviews, inspections, and test results and present it as a security case that allows system stakeholders to assess their degree of trust that the system protects its assets and preserves stakeholder privacy.

developing software for the military, government, or health agencies. Today, security must be the concern of every software engineer who has client resources to protect.

Quote:

Security is always excessive until it's not enough.

Robbie Sinclair

In the simplest sense, software security provides the mechanisms that enable a software system to protect its assets from attack. *Assets* are system resources that have value to one or more stakeholders. Assets include database information, files, programs, hard-drive storage space, system memory, or even processor capacity. *Attacks* often take advantage of software weaknesses or vulnerabilities that allow unauthorized access to a system. For example, a common vulnerability might come from failing to authenticate users before allowing them to access valuable system resources.

Software security is an aspect of software quality assurance (Chapter 21). Earlier in this book, you learned that quality cannot be added to a system by responding to bug reports. In a similar manner, it is very difficult to add security to an existing system by responding to reports of system vulnerabilities that have been exploited [Gho01]. Security concerns must be considered at the beginning of the software process, built into the software design, implemented as part of coding, and verified during testing and deployment.

In this chapter we present a survey of important issues in software security engineering. A complete discussion of this topic is beyond the scope of this book. Additional information may be found in [All08], [Lip10], and [Sin08].

27.1 Analyzing Security Requirements

ADVICE

Focus on assets that have the highest value and the greatest exposure.

Requirements for software security are determined by working with the customer to identify the assets that must be protected and the cost associated with the loss of those assets should a security breach occur. The value of the loss of an asset is known as its *exposure*. Losses might be measured in time or cost to recover or recreate an asset. Assets with insignificant value do not need to be secured.

An important part of building secure systems is anticipating conditions or threats that may be used to damage system resources or render them inaccessible to authorized users. This process is called *threat analysis*. Once the system assets, vulnerabilities, and threats have been identified, controls can be created to either avoid attacks or mitigate their damage.

Software security is an essential prerequisite for software integrity, availability, reliability, and safety (Chapter 19). It may not be possible to create a system that can defend its assets against all possible threats, and for that reason, it may be necessary to encourage users to maintain backup copies of critical data, redundant system components, and ensure privacy controls are in place.

SafeHome

Stakeholder Security Concerns

The scene: Software engineering team workspace.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member, Doug Miller, software engineering manager; and Lisa Perez, marketing team members, and a product engineering representative.

The conversation:

Vinod: If it's okay, I'll act as facilitator for this meeting.
(Everyone nods in agreement)

We need to begin determining the security concerns for the *SafeHome* Project.

Doug: Can we begin by listing the things we're worried about protecting?

Jamie: Well, what if an outsider hacks into *SafeHome* and manages to rob or damage a homeowner's house?

Lisa: The company's reputation would suffer if it was known some hacker disabled our systems.

Jamie: Not to mention the liability if the system was determined to be poorly designed.

Doug: The web interface to the product makes it possible for someone to intercept passwords as they're transmitted.

Ed: More importantly, the web interface will require a database containing customer information, so we have privacy concerns.

Vinod: Perhaps this would be a good time to have everyone spend 10 minutes listing each asset they think might be lost or compromised by an attack.
[10 minutes pass]

Vinod: OK, let's post them on the whiteboard and see if there are similar concerns.
[15 minutes and the list is created]

Lisa: That looks like a lot of concerns. How can we handle them all?

Doug: We need to prioritize our list based on the cost to repair the damage caused by losing the asset.

Lisa: How can we do that?

Vinod: We need to get real costs for replacing the lost assets using historic project data. And Lisa, you need to talk to legal and get their take on what our liability might be.

27.2 Security and Privacy in an Online World

Internet activity is moving away from traditional desktop browsing and toward scenarios in which browsers provide dynamic, customized content. Users can enter their own content on community forums that incorporate data mash-ups from third-party sites. Many desktop applications make use of web browser interfaces to access local data and provide a uniform user experience on multiple computing platforms. As a consequence, web developers need better control and security mechanisms. Data and code from untrusted sources should not be given the same privileges as a trusted programmer's code. For the web browser to be an effective user interface, privacy, trust, and security need to be among its most important quality attributes [Sei11].

Quote

"Relying on the government to protect your privacy is like asking a peeping tom to install your window blinds."

John Perry Barlow

In many instances a user's confidential information flows across organizational boundaries on the Internet. For example, electronic patient information may need to be shared among hospitals, insurance companies, and physicians. Similarly, travel information may need to be shared among travel agencies, hotels, and airlines. In these situations, users must reveal personal information to receive a service. Often the user is not able to control what an organization does

with this information once it is received in digital form. In ideal circumstances, users can directly control how their data should be handled and shared, but this requires the creation of user-prescribed data-sharing preferences to the data itself as it is captured electronically [Pea11].

27.2.1 Social Media

Quote:

"If you reveal your secrets to the wind, you should not blame the wind for revealing them to the trees."

Kahlil Gibran

The explosive growth and popularity of online social media networks have made them attractive targets for malicious programmers. Because of the implicit trust most users have in a social network environment, it is easy for a hacker to use a compromised account to send malware-infected messages to the account holder's friends. Social networks can be used to lure users to phishing sites[1] with the intention of tricking them into submitting personal data or forwarding cash to a friend in need. Another ruse is the use of e-mails that include details stolen from a user's personal data [Sae11].

Some social media networks allow users to develop their own applications. These applications innocently ask the user to grant access to personal data and then use that data in ways that the user never intended. Sometimes an application or game is so attractive that even a knowledgeable user will allow this type of access so that she can use the application. Social networks often have check-in features that allow criminals to target individuals through their movements in real life. Despite the concerns over identity theft, spam, and spyware, many computer users are not motivated to take the steps needed to protect themselves [Sta10].

27.2.2 Mobile Applications

? What threats are encountered in mobile applications?

Users of mobile apps have access to almost the same web services that fixed-location wired Internet users have. Wireless Internet users have inherited all the security risks associated with desktop commerce, plus many new risks unique to mobile networks. The nature of wireless networks requires trust and cooperation between nodes that can be exploited by malicious programs to deny services or collect confidential information. The platforms and languages developed for mobile devices can be hacked, and malicious code can be inserted into the device system software with all the privileges of the device owner. This means that security technologies such as sign-in, authentication, and encryption can be undermined with relative ease.

27.2.3 Cloud Computing

Cloud computing brings additional confidentiality and privacy concerns because data is entrusted to remote servers that are managed by service providers. These cloud service providers have complete access and control over our data. They are trusted not to share it with others (either accidentally or deliberately)

1 A *phishing site* disguises itself as a known and trusted website, luring the user to provide personal information that can lead to the loss of security assets.

and to take responsible steps to prevent its loss. Online data repositories are very attractive sources for data mining (e.g., to collect demographic or marketing information). The problem is that the originator of the data has not given his explicit consent [Rya11a]. Policy makers should create policies and regulations to ensure that service providers do not abuse the trust of their users.

The boundary between the "trusted inside" and "untrusted outside" blurs when a company adopts cloud computing. With the organization's applications and data no longer on site, a new type of malicious insider is possible. Confidential data are only a few commands away from access by a malicious or incompetent system administrator. Most cloud service providers have strict procedures in place for monitoring employee access to customer data. Policies preventing physical access to data are not effective against remote attacks, and monitoring often only detects an attack after it happens. To foster user trust in a cloud system, it may be important to provide users with the ability to assess whether the necessary mechanisms for protecting confidentiality and privacy are in place [Roc11].

The ubiquity of Web access and the advent of cloud computing has enabled new forms of business collaborations. Sharing information and protecting confidentiality is a difficult task. Secure multiparty computing raises the risks of selfish behavior unless all parties are confident no one can take advantage of the systems. This situation highlights a psychological dimension of system trust and security that cannot be solved by software engineering alone [Ker11].

27.2.4 The Internet of Things

Some visionaries describe an "Internet of Things" [Rom11] in which everything real has a virtual counterpart on the Internet. These virtual entities can produce and consume services and collaborate toward a common goal. For example, a user's phone knows about the user's physical and mental state through a network of surrounding devices that can act on the user's behalf. Automotive engineers envision cars that communicate autonomously with other vehicles, data sources, and devices and do not require direct driver control.

However, security is one of the major obstacles that stand in the way of this vision. Without strong security foundations, attacks and malfunctions will outweigh any of the benefits realized by an Internet of things. Policy makers must consider the balance between governance and innovation. Excessive governance can easily hinder innovation, and in turn, innovation can inadvertently ignore human rights [Rom11].

27.3 Security Engineering Analysis

Security analysis tasks include requirements elicitation, threats modeling, risk analysis, measures design, and correctness checks. These tasks include consideration of both the functional and nonfunctional details of the system along with its business case [Bre03].

27.3.1 Security Requirement Elicitation

Quote:

"The user's going to pick dancing pigs over security every time."

Bruce Schneier

The general techniques for requirement elicitation discussed in Chapter 8 are equally applicable to the elicitation of security requirements. Security requirements are nonfunctional requirements[2] that strongly influence the *architectural design* of software systems. Once the system requirements have been refined and prioritized using threat modeling and risk analysis, security policies can be set for the system. These policies will be refined using security modeling and decomposition along with usage considerations to derive the required security architecture. The security aspects of the architecture are validated before they are implemented [Bod09].

In some cases, security requirements and other software requirements can come into conflict with one another. For example, security and usability can be at odds with one another, and a balance between the two must be found. Highly secure systems are often harder to use by inexperienced users. In *user-centered security engineering*, security requirements elicitation finds answers to three important questions [Mar02]: (1) What are the users' needs with respect to security software? (2) How can a secure architecture be designed so that it accommodates a good user interface design? (3) How can a good user interface be designed so that it is secure but at the same time enables effective, efficient, and satisfying usage? The answers to these questions should be incorporated into the use case scenarios (Chapter 8) describing how stakeholders interact with the system resources.

? What security requirement elicitation questions should I ask?

As requirement elicitation proceeds, the analyst should identify attack patterns. An *attack pattern* is a type of design pattern (Chapter 16) that identifies the security shortcoming of a system. Attack patterns can speed up security analysis by providing problem/solution pairs to common security vulnerability. Reusing attack patterns can help engineers to identify system vulnerabilities. There is no need to recreate different ways to attack a system. Attack patterns allow developers to use well-understood names (e.g., phishing, SQL injection, and cross-site scripting) for software security issues. Common attack patterns can be improved over time [Sin08]. The difficulty with using attack patterns is knowing when a specific pattern will apply.

Quote:

"Securing a computer system has traditionally been a battle of wits: the penetrator tries to find the holes, and the designer tries to close them."

Gosser

Some software engineers believe that the rigors of security engineering are incompatible with the informal nature of requirements elicitation in agile processes (Chapter 5). However, one technique that might be used to reconcile the "formality gap" is the creation of abuser stories in the requirements domain. *Abuser stories* are based on customer input that describes threats to the system assets. Abuser stories extend the well-established agile concept of user stories and can help achieve the security requirements traceability needed to allow security assurance to proceed [Pee11].

2 These are sometimes called *crosscutting concerns* and were discussed in Chapter 4.

27.3.2 Security Modeling

Modeling is an important process for specifying and analyzing requirements. A *security model* is a formal description of software system *security policy.* A security policy provides a definition of system security that captures its key security requirements and also contains rules describing how security is to be enforced during system operation.

The security model can provide precise guidance during the design, coding, and review processes. Once the system is built, the security model provides a basis to aid in verifying the correctness of the security implementation [Dan09]. The security model is also a valuable security reference as the system evolves or requires repair during maintenance activities.

? What information is contained in a security model?

A security model may be represented using text or graphical formalisms. Regardless of their representation a security model needs to capture the following items: (1) security policy objectives, (2) external interface requirements, (3) software security requirements, (4) rules of operation, and (5) specifications describing model-system correspondence.

Some security models are represented on state machines.[3] Each state must include information on the relevant security aspects of the system. As a software engineer who is concerned with security, you must be certain that any state transitions allowed in the system start in a secure state and end in a secure state. You must also verify that the initial system state is a secure state. To be complete, the model needs to have an interpretation that shows how it relates to the actual system.

Executable modeling formalisms allow developers to verify a security model and its behavior before accepting it. Once accepted, the model forms a good basis for design. Two languages used for modeling security requirements are *UML.sec* (an extension to UML using stereotypes and constraints) and *GRL* (goal-oriented requirements language for capturing nonfunctional requirements). The use of formal modeling languages may help increase the trustworthiness of the system as it is developed [Sal11].

Formal methods (Chapter 28) have been proposed as a means of augmenting the security analysis and verification of systems. Use of formal specifications for the security requirements has the potential to assist in the creation of test cases for model-based security testing. The use of formal correctness proofs for critical system components can add to the developers' confidence that the system does indeed conform to its specification. Of course care must be taken that the assumptions underlying the proofs are satisfied.

3 A *finite state machine* is defined by a list of the possible transition states from each current state, and the triggering condition for each transition.

27.3.3 Measures Design

KEY POINT

Secure software must exhibit three properties: dependability, trustworthiness, and survivability.

To be secure, software must exhibit three properties: *dependability* (operates under hostile conditions), *trustworthiness* (system will not behave in a malicious manner), and *survivability* (continues to operate when compromised).[4] Security metrics[5] and measures need to focus on assessing these properties.

Useful security metrics must be based on measures that allow developers to assess the degrees to which data confidentiality or system integrity may be at risk. Three measurements that are needed to create such metrics are measures of *asset value, threat likelihood,* and *system vulnerability*. These properties are not easy to measure directly. The cost of losing an asset may be more than the cost of recreating it.

The best measures are those that are readily available during the development or operation of the software. The number of desk security complaints or number of security test cases failed can provide some measures (e.g., the number of identity theft incidents reported each month). Vulnerabilities may not be known until attacks occur, but the number of successful attacks can be counted.

27.3.4 Correctness Checks

Quote:

"In theory, one can build provably secure systems. In theory, theory can be applied to practice but in practice, it can't."

M. Dacie

Security correctness checks need to take place throughout the software development cycle. The exposure to stakeholder assets from attacks on system vulnerabilities should be determined early in the development process.

The software team then ensures that the threat model derived from the system use cases has been accounted for in the security portions of the risk mitigation, monitoring, and management plan. Quality assurance activities include the identification of security standards and the development of security guidelines for use during modeling and construction activities. Software verification activities ensure that the security test cases are complete and traceable to the system security requirements.

Many of these security checks should be included in the audits, inspections, and testing activities built into the conventional software engineering tasks (Section 27.6). Data collected during these checks is analyzed and summarized as part of the system *security case* as described in Section 27.4. The *trust verification* process is discussed in Section 27.7.

---

4 **https://buildsecurityin.us-cert.gov**

5 *Metrics* are a quantitative indicator of the degree to which a system component or processes possess a given attribute. Good metrics meet the SMART (i.e., specific, measurable, attainable, repeatable, and time-dependent) criteria. Metrics are typically derived from measurements using statistical techniques to uncover the relationships. Additional discussion on metrics and measurements appears in Chapter 30.

27.4 Security Assurance

Because software has become integrated into our everyday lives, security flaws and the losses associated with them become more costly and more ominous. Sound software engineering practice involves ascertaining requirements, developing an appropriate design, and demonstrating that you have created the right product. Security assurance is intended to demonstrate that you have built a secure product, and as a consequence, it inspires confidence among end users and other stakeholders.

27.4.1 The Security Assurance Process

Verification is the part of the assurance task that provides evidence that stakeholders can be confident that the software conforms to requirements. When considered in the context of security engineering, you choose a critical subset of the security requirements or claims for the software and create an assurance case that demonstrates that the software conforms to those requirements or claims.

KEY POINT

A security case supports the claim that the software is secure.

An *assurance case* is a reasoned and auditable artifact that supports the contention that the software satisfies the claim being asserted. Assurance cases have long been used for software safety and are now being used for software security; they are often called *security cases*.

Each security case consists of three elements: (1) the claims themselves, (2) the arguments that link the claims to each other through evidence and assumptions, and (3) the body of evidence and explicit assumptions that support the arguments.

To be valid a security case must satisfy three objectives. It must specify claims that are suitable and adequate for the system in question, document that suitable engineering practices have been applied so that the claims can be achieved, and show the achievement of these claims are within the required level of risk [Red10].

Several types of evidence may be used to prove the security case. Formal proofs of software correctness (Chapter 28) may be helpful, if the code was designed with the intension of proving its correctness. There are tools that support automatic software verification [DSi08] and other tools that perform static scans for software security weaknesses (e.g., RATS, ITS4, SLAM).[6] But tools by themselves cannot build a security case.

Some of the evidence will come from the review of system artifacts using variations of formal technical reviews and inspections of the artifacts (Chapter 20). However, these reviews focus solely on the security claims. Reviewers with security expertise might review the system or its security case. Checklist evaluations might also be used to verify that security guidelines and process steps were followed.

6 For a list of security tools, see: **http://www.tech-faq.com/how-to-find-security-vulnerabilities-in-source-code.html**

SafeHome

Building the Security Case

The scene: Software engineering team workspace.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member, Bridget Thornton, software quality group leader.

The conversation:

Ed: Thanks for joining us Bridget, we need to build the security case for *SafeHome* project.

Vinod: How should we get started?

Bridget: We might start by picking one security concern and see what evidence we can find to support the case for it.

Ed: What kind of evidence?

Bridget: Let's pick one of the security concerns first.

Vinod: Let's focus on security concerns related to the customer database.

Bridget: OK, let's start by listing the security claims made for accessing the database.

Jamie: Do you mean the security model elements that refer to the database?

Bridget: Yes. Next, we take a look at the completed inspection checklists and the summaries of the formal technical reviews that have been happening as each project milestone is completed.

Ed: What about the process audits and change request documents produced by your group?

Bridget: Those are important to include as well.

Vinod: We used an ITG to create and run most of the system test cases.

Bridget: A summary of the behavior of the security test cases comparing expected and actual output is a very important part of the security case.

Jamie: That seems like a lot of information to get a handle on.

Bridget: It is. That's why the next step is to take each claim made for database security and summarize the evidence supporting or refuting the claim of adequate asset protection.

Ed: Can you help us review our security case when it's assembled?

Bridget: Of course. My group needs to have an ongoing dialog with your team as this project moves forward, both pre- and post-launch.

27.4.2 Organization and Management

The rush to get software out the door quickly often forces project managers to be more concerned with features and functions than security. Software engineers should focus on robustness in software design and architecture, but, in addition, secure practices should be adopted as software-based systems are built [Sob10].

Security assurance and risk identification activities are planned, managed, and tracked in the same way as other software engineering activities. The software team collects data (e.g., number of access violations, number of system failures, number of data records lost) to determine what works and what does not. This requires developers to analyze each reported failure (to determine whether its cause is related to a system vulnerability) and then assess the asset exposure that resulted.

27.5 Security Risk Analysis[7]

KEY POINT

Threat modeling is a security analysis method that can be used to identify those threats with the highest potential to cause damage.

Identifying and managing security risks are important project planning tasks (Chapter 31). Security engineering is driven by risks that are identified by the software team and other stakeholders. Risks impact project management and security assurance activities.

Threat modeling is a security analysis method that can be used to identify those threats with the highest potential to cause damage to a software-based system. Threat modeling is accomplished in the earliest phases of a project using the requirements and analysis models. The creation of a threat model involves identifying key components of an application, decomposing the application, identifying and categorizing the component threats, rating and categorizing the threats to each component, rating the components based on their risk ranking, and developing risk mitigation strategies. Microsoft uses the following steps to create a threat model [Sin08]:

? What are the steps required to build a threat model?

1. **Identify assets**. List all sensitive information and intellectual property, where it is stored, how it is stored, and who has access.
2. **Create an architecture overview.** Write system use cases and build a model of the system components.
3. **Decompose the application.** The goal is to ensure that all data sent among application components are validated.
4. **Identify threats.** Note all threats that might compromise the system assets using methods like attack trees or attack patterns, often the process involves looking as network, host system configuration, and application threats.
5. **Document the threats.** Create a risk information sheet detailing how each threat should be monitored and mitigated.
6. **Rate the threats**. Most projects have insufficient resources to address all conceivable threats, so threats need to be ranked using their impact and likelihood.

Costly assets should be protected from high probability risks. A quantitative risk-assessment process (Chapter 35) can be used to rank the threats. First, all assets to be assessed are identified and the dollar value of losing or recreating an asset is determined. For each asset, a list of major threats is created and historical data are used to determine the likely occurrence of each threat in a typical year. The potential loss in dollars per major threat for each asset per year is computed along with the *annual loss expectancy* (ALE) (determined by multiplying the occurrence and potential loss). Finally, the combined threat of losing an asset is computed by adding the ALE values associated with each individual threat.

7 A generic discussion of risk analysis for software projects encompassing all types of risks that threaten the project and its success is presented in Chapter 35.

SafeHome

Security Steps

The scene: Software quality assurance group workspace.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Bridget Thornton, software quality group leader.

The conversation:

Vinod: Hi Bridget. Doug wants us to work on security risk analysis.

Bridget: Is this to help set the security priorities for development?

Jamie: I think so.

Vinod: Can we look at database security concerns?

Bridget: Sure. We know what the costs are to back up and repair the data records using historical data. We may not know the liability damages that might be awarded if customer data is stolen, but we have industry data on those costs.

Jamie: Is that all we need?

Bridget: Well, you already have the system architectural diagrams. It's easier to verify that all data exchanged among the components have been validated. We'll also need to determine the threats to each asset.

Vinod: How do we do that?

Bridget: We might create an attack tree. We would start by setting an attack goal at the root. For example, an attacker's goal might be to steal customer information.

Vinod: And . . .

Bridget: You then look at your database attack pattern catalog to see which apply and list each as subgoals in the tree.

Jamie: Then what?

Bridget: You need to refine the threats and create risk information sheets for each, describing impact of the threat and any monitoring or mitigation steps that should be in place to address it.

Vinod: How does this help set development priorities?

Bridget: You determine the cost of each threat by computing the annual loss expectancy (ALE) for each threat using historical data. We can help you with that part of the process.

Jamie: Thanks, Bridget. We'll be back to get your input on that ALE computation once we have the threats identified and refined.

27.6 The Role of Conventional Software Engineering Activities

Quote:

In a world in which the total of human knowledge is doubling about every 10 years, our security can rest only on our ability to learn.

Nathaniel Branden

Building a system and making it secure *after the fact* is highly inefficient and prone to failure. Yet, some software developers argue that threats to a system cannot be predicted until it is built. As a consequence, they ignore security issues until the testing phase and then "back-fill" to eliminate security mistakes that were made earlier in the software process. Adding security patches to an existing system in an ad hoc manner may not be possible without major changes to the design or architecture of the system. Hence, the back-fill approach is both inefficient and costly.

The nature of an iterative and incremental process (Chapter 4) makes it difficult to address all security concerns before doing any development work. Software requirements often change during the development process. In addition, architectural design decisions can have a direct impact on security concerns. For these reasons, it is difficult to address all security issues at the beginning of a project. Even when the majority of security concerns are addressed up front,

design decisions later in the software process can affect security vulnerabilities in the final system [Mei06].

An effective software process includes a reasonable set of review and adjustment opportunities. Many security activities complement one another and have a synergistic effect on software quality. For example, code reviews are known to reduce the number of product defects prior to testing, which in turn eliminates potential security holes and improves software quality.

During *planning* the project budget and timeline must take security into account so that the resources needed to satisfy the security objectives[8] of the system are appropriately allocated. As part of the security and privacy risk assessment, each functional requirement needs to be examined to see if it can affect an asset that is associated with a system security objective. During risk analysis, the cost associated with each loss should be determined or estimated.

KEY POINT

An *attack surface* is defined as the set of reachable and exploitable vulnerabilities present in a software product.

The identification of mechanisms for dealing with specific threats to the system is often delayed until the requirements for a software increment are translated to its *design* requirements. This is where the attack surface should be identified. The *attack surface* is defined as the set of reachable and exploitable vulnerabilities present in a software product. Many security vulnerabilities will be found at the intersections of the system layers. For example, information entered via a form in the user interface may be intercepted as it travels across a network to a database server. Design guidelines that include security provisions that directly address the attack surface can be developed.

It may be helpful to separate the security reviews from the general design reviews. Code reviews that focus on security issues should be included as part of the *implementation* activities. These code reviews should be based on the appropriate security objectives and threats identified in the system design activities.

? What is an incident response plan?

Security testing is a routine part of system testing (Chapter 22). Security risk assessment can be a source of test cases that enable security testing to become more focused. An incident response plan (IRP) spells out the actions to be carried out by each system stakeholder in response to specific attacks [Pra07]. A thorough review of the IRP should also be part of the security verification process.

Quote:

"When you know that you're capable of dealing with whatever comes, you have the only security the world has to offer."

Harry Browne

In addition, verification should include separate reviews of security operations and asset archiving procedures. The security risk management plan should be reviewed periodically as part of the maintenance process.

When security incidents are reported after an application has been deployed, developers should evaluate the effectiveness of the security risk management procedures as part of the system maintenance (Chapter 36). If the system change procedures (Chapter 29) include root cause analysis, this may help uncover vulnerabilities in the overall system design.

8 For example, protection of customer data or recognition of legal or regulatory compliance requirements associated with the confidentiality, integrity, or availability of the system information or other intellectual property.

27.7 Verification of Trustworthy Systems

KEY POINT

Trust indicates the level of confidence that one system entity (or organization) can rely on another.

When considered in the context of software security, *trust* indicates the level of confidence that one system entity (or organization) can rely on another. *System entities* encompass entire systems, subsystems, and software components. Trust has a psychological dimension as well as a technical dimension. In general, one entity can be said to trust another when it makes the assumption that the second entity will behave exactly as the first entity expects. Demonstrating that this assumption is correct is the task of verifying the trustworthiness of a system. Although a variety of trust models have been proposed [Sin08], our focus will be on ensuring that the system conforms to the mitigation practices created within the system threat model.

The verification task ensures that the requirements for trustworthy systems are assessed using specific, quantifiable metrics based on testing, inspection, and analysis techniques [She10]. Testing metrics might include the ratio of the number of detected faults to the number of predicted faults or the ratio of security test cases passed to the total number run. Other metrics might include defect removal efficiency (Chapter 32) of the formal inspection activities. Ensuring the traceability of security test cases back to security use cases developed during analysis activities is also useful.

The evidence used to prove the security case must be acceptable and convincing to all collaborators of the trusted entity. Users of trustworthy systems should be convinced that the system has no exploitable vulnerabilities or malicious logic. As a consequence of the verification task, users should have confidence in the dependability of the system and its survivability when compromised. This implies that damage to software will be minimized and that the system can recover quickly to an acceptable operating capacity. Specific security test cases and procedures are an important part of the verification process as well [Mea10].

SafeHome

Security Test Case Creation

The scene: Vinod's cubical.

The players: Vinod Raman, software team member, and Ed Robbins, software team member.

The conversation:

Vinod: We need to create a security test case for accessing the *SafeHome* video offsite.

Ed: We should start by reviewing the security use case that Doug and Bridget [software quality group leader] developed.

Vinod: I suppose we could let the ITG contractors do this, but this seems like a pretty straightforward test case. It should be added to the set of test cases we use for regression testing, too.

Ed: Okay, the password use case calls for the user to log on to a website using a secure connection with a valid user ID, two levels of passwords, and the user to enter a four-digit pin after requesting the video feed request.

Vinod: That gives us several logic paths to test. There are four pieces of data for the user to enter. Each input needs to be tested with a good value, an incorrect value, a null value, and an incorrectly formatted data value.

Ed: To cover all logic paths requires 256 distinct test cases.

Vinod: Yes, it does. We also need to define the response for each.

Ed: Based on the security policy, the user has three attempts for each piece of information.

Vinod: Right, and the user is prompted to enter the data after each bad attempt.

Ed: And if any one of them fails on the third attempt the system is supposed to send an e-mail alert to the company and the user.

Vinod: It would probably be good to randomize the order the test cases are presented to the password checker. We might need to run our test cases more than once to be confident the password checker is not history sensitive.

Ed: We should write a small program to run through these test cases and log the results.

Vinod: Yeah, this is a lot of work. Maybe we should let the ITG work with Bridget's SQA team to develop the security tests.

Today, software quality measurement does not adequately address trust assurance and security. Existing measures (e.g., reliability measures like mean time between failure or dependability measures such as defect density) often overlook numerous factors that may compromise software and leave it vulnerable to attack. In part, this is because many of these metrics do not take into account the fact that there are active agents continually probing the software for vulnerabilities.

Effective security metrics maintain historical data based on an entity's past behavior in situations involving trust. As an example, consider the trust established when an e-commerce site allows rating of their sellers and buyers. Of course, this type of rating system must ensure that entities being rated are correctly identified and do not have inaccurate data recorded about the entity. Problems like these sometimes plague credit reporting systems.

The U.S. Department of Homeland Security advocates the adoption of secure software design practices that employ reliable, standardized measurement tools. Ideally these tools (when they exist) can help developers reduce the number of vulnerabilities introduced into a system during development [Mea10]. This might allow users of trusted systems to make informed decisions about the trustworthiness of a system. But like system reliability, the user may base this judgment on the extent of losses experienced using the system.

SOFTWARE TOOLS

Security Engineering

Objective: Security engineering tools assist in identifying security vulnerabilities in source code.

Mechanics: In general the software source code is processed by allowing the tool to read the source code and flag programming constructs for developers to examine carefully.

Representative Tools:[9]

RATS (Rough Auditing Tool for Security), developed by Secure Software (**http://code.google.com/p/rough-auditing-tool-for-security/**), is a scanning tool that provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggesting remedies.

9 Tools noted here do not represent an endorsement, but rather are a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

ITS4, developed by Cigital (**http://freecode.com/projects/its4/**), is a tool for statically scanning security critical C and C++ source code for vulnerabilities.

SLAM, developed by Microsoft (**http://research.microsoft.com/en-us/projects/slam/**), is a tool for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct functioning.

Many additional security source code–scanning tools can be found at **http://www.tech-faq.com/how-to-find-security-vulnerabilities-in-source-code.html**

27.8 Summary

Software security engineering is concerned with developing software that protects the assets it manages from threats. Threats may involve attacks that exploit system vulnerabilities to compromise the confidentiality, integrity, or availability of system services or data.

Security risk management is concerned with assessing the impact of possible threats and deriving security requirements to minimize critical losses. Design for security involves creating a system architecture that minimizes the introduction of known vulnerabilities. Software engineers should make use of techniques to prevent attacks, to repel attacks, and to recover from attacks as a means to mitigating the effects of losses.

Inspiring trust among stakeholders requires that developers regard security assurance as an umbrella activity that is present at the beginning throughout the software process. The development of security metrics is still in its infancy. Building a security case for a system involves collecting evidence using security testing, conducting security-focused formal technical reviews, and inspecting to ensure security guidelines and mitigation practices are being followed.

Problems and Points to Ponder

27.1. Consider a mobile phone app that you own. First describe the app briefly and then list at least three to five security risks.

27.2. Describe a security migration strategy for one of the risks noted in Problem 27.1.

27.3. Identify five attack patterns that may be commonly used to attack web apps.

27.4. Describe the trust model used on a bidding site such as eBay.

27.5. Describe the security requirements for a cloud-based photo repository.

27.6. What does the same origin policy have to do with trustworthy systems?

27.7. Use the Internet to determine the annual average cost of single incidence of identity theft to an individual consumer.

27.8. Explain some of the problems that might be encountered if you try to address security risk after a system is completed.

27.9. Use the Internet to find the details needed to create a phishing attack pattern.

27.10. Compute the annual loss expectancy (ALE) for the loss of a data server whose replacement value is $30,000, the occurrence of loss of data due to hacking is 5 percent annually, and the potential loss is $20,000.

Further Readings and Information Sources

Books by Vacca (*Computer and Information Security Handbook,* 2nd ed., Morgan Kaufman, 2013), Goodrich and Tamassia (*Introduction to Computer Security,* Addison-Wesley, 2010), Anderson (*Security Engineering,* 2nd ed., 2008), Kern et al. (*Foundations of Security,* Apress, 2007), McGraw (*Software Security,* Addison-Wesley, 2006), and Dowd and his colleagues (*The Art of Software Assessment: Identifying and Preventing Software Vulnerabilities,* Addison-Wesley, 2006) address important security issues. Zalewski (*The Tangled Web: A Guide to Securing Modern Web Applications,* No Starch Press, 2011) and Howard and LeBlanc (*Writing Secure Code,* 2nd ed., Microsoft Press, 2003) discuss how to construct secure systems. A project manager's view is presented by Allen et al. (*Software Security Engineering,* Addison-Wesley, 2008). Howard and Lipner (*The Security Development Lifecycle,* Microsoft Press, 2006) discuss the security engineering process. Schumacher et al. (*Security Patterns,* Wiley, 2006) provide insight into the use of patterns as one element of effective security engineering.

Others focus on the "break-in" side. Books by Barnett and Grossan (*Web Application Defender's Cookbook: Battling Hackers and Protecting Users,* Wiley, 2012), Ottenheimer and Wallace (*Securing the Virtual Environment: How to Defend Against Enterprise Attack* , Wiley, 2012), Studdard and Pinto (*The Web Application Hacker's Handbook,* Wiley, 2011), Howard and his colleagues (*24 Deadly Sins of Software Security,* McGraw-Hill, 2009), Erickson (*Hacking: The Art of Exploration,* No Starch Press, 2008), Andrews and Whittaker (*How to Break Web Software,* Addison-Wesley, 2006), Whittaker (*How to Break Software Security,* Addison-Wesley, 2003), provide useful insight for software engineers who need to understand how to attack systems and applications.

Sikorski and Honig (*Practical Malware Analysis,* No Starch Press, 2012) and Ligh et al. (*Malware Analyst's Cookbook,* Wiley, 2010) provide excellent insight into the inner working of malware. Swiderski (*Threat Modeling,* Microsoft Press, 2004) presents a detailed discussion of threat modeling.

Guidelines for those who must conduct security testing are provided in books by Allen (*Advanced Penetration Testing for Highly Secured Environments,* Packt Publishing, 2012), O'Gorman (*Metaspoit: The Penetration Tester's Guide,* No Starch Press, 2011), Faircloth (*Penetration Tester's Open Source Tool Kit,* Syngress, 2011), Engebretson (*The Basics of Hacking and Penetration Testing,* Syngress, 2011), Faircloth (*Penetration Tester's Open Source Tool Kit,* Syngress, 2011), Hope and Walther (*Web Security Testing Cookbook,* O'Reilly Media, 2008), and Wysopal et al. (*The Art of Software Security Testing,* Addison-Wesley, 2006).

A wide variety of information sources on security engineering are available on the Internet. An up-to-date list of World Wide Web references that are relevant to pattern-based design can be found at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

28

Formal Modeling and Verification

Key Concepts

box structure specification 604
certification 612
cleanroom design . 607
cleanroom process model. 603
correctness verification. 608
design refinement 608

Unlike reviews and testing that begin once software models and code have been developed, formal modeling and verification incorporate specialized modeling methods that are integrated with prescribed verification approaches. Without the appropriate modeling approach, verification cannot be accomplished.

In this chapter and in Appendix 3 we discuss two formal modeling and verification methods—*cleanroom software engineering* and *formal methods*. Both demand a specialized specification approach and each applies a unique verification method. Both are quite rigorous and neither is used widely by the software engineering community. But if you intend to build bulletproof software, these methods can help you immeasurably. They're worth learning about.

Quick Look

What is it? How many times have you heard someone say, "Do it right the first time"? If we achieved that in software, there'd be considerably less effort expended on unnecessary software rework. Two advanced software engineering methods—cleanroom software engineering and formal methods—help a software team to "do it right the first time" by providing a mathematically based approach to program modeling and the ability to verify that the model is correct. Cleanroom software engineering emphasizes mathematical verification of correctness before program construction commences and certification of software reliability as part of the testing activity. Formal methods use set theory and logic notation to create a clear statement of facts (requirements) that can be analyzed to improve (or even prove) correctness and consistency. The bottom line for both methods is the creation of software with extremely low failure rates.

Who does it? A specially trained software engineer.

Why is it important? Mistakes create rework. Rework takes time and increases costs. Wouldn't it be nice if you could dramatically reduce the number of mistakes (bugs) introduced as the software is designed and built? That's the premise of formal modeling and verification.

What are the steps? Requirements and design models are created using specialized notation that is amenable to mathematical verification. Cleanroom software engineering uses box structure representation that encapsulates the system (or some aspect of the system) at a specific level of abstraction. Correctness verification is applied once the box structure design is complete. Once correctness has been verified for each box structure, statistical usage testing commences. Formal methods translate software requirements into a more formal representation by applying the notation and heuristics of sets to define the data invariant, states, and operations for a system function.

What is the work product? A specialized, formal model of requirements is developed. The results of correctness proofs and statistical use tests are recorded.

How do I ensure that I've done it right? Formal proof of correctness is applied to the requirements model. Statistical use testing exercises usage scenarios to ensure that errors in user functionality are uncovered and corrected.

design verification 608
formal methods. . . 615
functional specification 604
statistical use testing 610

Cleanroom software engineering is an approach that emphasizes the need to build correctness into software as it is being developed. Instead of the classic analysis, design, code, test, and debug cycle, the cleanroom approach suggests a different point of view [Lin94]:

> The philosophy behind cleanroom software engineering is to avoid dependence on costly defect removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system.

In many ways, the cleanroom approach elevates software engineering to another level by emphasizing the need to *prove* correctness.

Models developed using *formal methods* are described using a formal syntax and semantics that specify system function and behavior. The specification is mathematical in form (e.g., predicate calculus can be used as the basis for a formal specification language). In his introduction to formal methods, Anthony Hall [Hal90] makes a comment that applies equally to cleanroom methods:

> Formal methods [and cleanroom software engineering] are controversial. Their advocates claim that they can revolutionize [software] development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods [and cleanroom software engineering] are so unfamiliar that it is difficult to judge the competing claims.

We present an overview of formal modeling and verification concepts in this chapter. In Appendix 3 we explore some of the technical details of formal modeling and verification.

28.1 The Cleanroom Strategy

Cleanroom software engineering makes use of a specialized version of the incremental software model introduced in Chapter 4. A "pipeline of software increments" [Lin94] is developed by small independent software teams. As each increment is certified, it is integrated into the whole. Hence, functionality of the system grows with time.

The sequence of cleanroom tasks for each increment is illustrated in Figure 28.1. Within the pipeline for cleanroom increments, the following tasks occur:

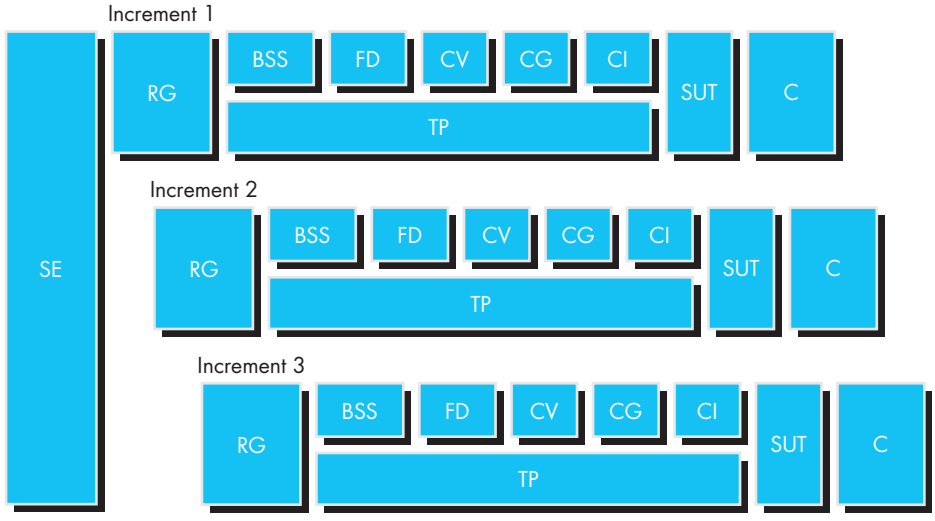
? What is the process model that is used for cleanroom projects?

Increment planning. A project plan that adopts the incremental strategy is developed. The functionality of each increment, its projected size, and a cleanroom development schedule are created. Special care must be taken to ensure that certified increments will be integrated in a timely manner.

Requirements gathering. Using techniques similar to those introduced in Chapter 8, a more-detailed description of customer-level requirements (for each increment) is developed.

FIGURE 28.1

The cleanroom process model



SE — system engineering
RG — requirements gathering
BSS — box structure specification
FD — formal design
CV — correctness verification
CG — code generation
CI — code inspection
SUT — statistical use testing
C — certification
TP — test planning

Quote:

"The only way for errors to occur in a program is by being put there by the author. No other mechanism is known . . . Right practice aims at presenting insertion of errors and, failing that, removing them before testing or any other running of the program."

Harlan Mills

Box structure specification. A specification method that makes use of *box structures* is used to describe the functional specification. Box structures "isolate and separate the creative definition of behavior, data, and procedures at each level of refinement" [Hev93].

Formal design. Using the box structure approach, cleanroom design is a natural and seamless extension of specification. Although it is possible to make a clear distinction between the two activities, specifications (called *black boxes*) are iteratively refined (within an increment) to become analogous to architectural and component-level designs (called *state boxes* and *clear boxes,* respectively).

Correctness verification. The cleanroom team conducts a series of rigorous correctness verification activities on the design and then the code. Verification (Section 28.3.2) begins with the highest-level box structure (specification) and moves toward design detail and code. The first level of correctness verification occurs by applying a set of "correctness questions" [Lin88]. If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

Code generation, inspection, and verification. The box structure specifications, represented in a specialized language, are translated into the appropriate programming language. Technical reviews (Chapter 20) are then used to ensure semantic conformance of the code and box structures

and syntactic correctness of the code. Then correctness verification is conducted for the source code.

Statistical test planning. The projected usage of the software is analyzed, and a suite of test cases that exercise a "probability distribution" of usage is planned and designed (Section 28.4). Referring to Figure 28.1, this cleanroom activity is conducted in parallel with specification, verification, and code generation.

ADVICE

Cleanroom emphasizes tests that exercise the way software is really used. Use cases provide input to the test planning process.

Statistical use testing. Recalling that exhaustive testing of computer software is impossible (Chapter 23), it is always necessary to design a finite number of test cases. Statistical use techniques [Poo88] execute a series of tests derived from a statistical sample (the probability distribution noted earlier) of all possible program executions by all users from a targeted population (Section 28.4).

Certification. Once verification, inspection, and usage testing have been completed (and all errors are corrected), the increment is certified as ready for integration.

The first four activities in the cleanroom process set the stage for the formal verification activities that follow. For this reason, we begin the discussion of the cleanroom approach with the modeling activities that are essential for formal verification to be applied.

28.2 Functional Specification

KEY POINT

A "box" encapsulates the system at some level of abstraction and is used in functional specification.

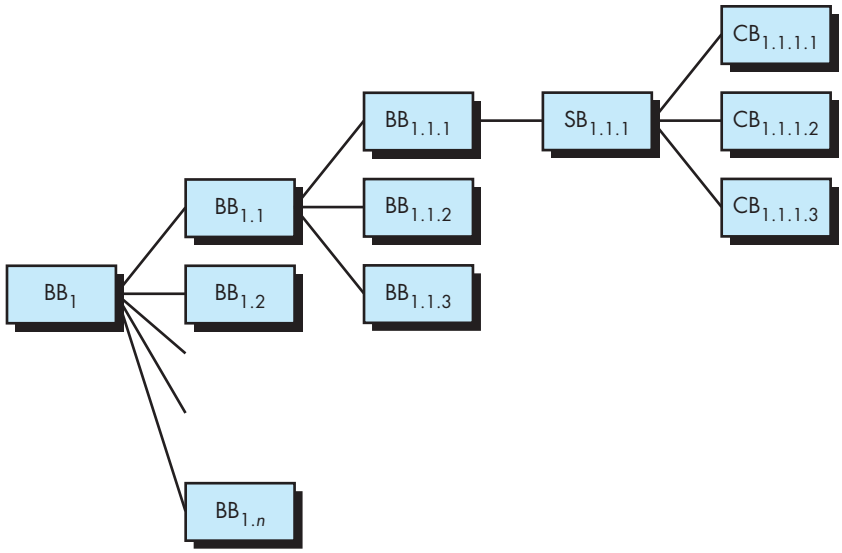
The modeling approach in cleanroom software engineering uses a method called *box structure specification.* A "box" encapsulates the system (or some aspect of the system) at some level of detail. Through a process of elaboration or stepwise refinement, boxes are refined into a hierarchy where each box has referential transparency. That is, "the information content of each box specification is sufficient to define its refinement, without depending on the implementation of any other box" [Lin94]. This enables the analyst to partition a system hierarchically, moving from essential representation at the top to implementation-specific detail at the bottom. Three types of boxes are used:

Black box. The black box specifies the behavior of a system or a part of a system. The system (or part) responds to specific stimuli (events) by applying a set of transition rules that map the stimulus into a response.

State box. The state box encapsulates state data and services (operations) in a manner that is analogous to objects. In this specification view, inputs to the state box (stimuli) and outputs (responses) are represented. The state box also represents the "stimulus history" of the black box, that is,

FIGURE 28.2

Box structure refinement



the data encapsulated in the state box that must be retained between the transitions implied.

Clear box. The transition functions that are implied by the state box are defined in the clear box. Stated simply, a clear box contains the procedural design for the state box.

Quote:

"Every solution to every problem is simple. It's the distance between the two where the mystery lies."

Derek Landy

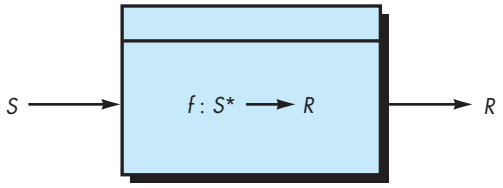
Figure 28.2 illustrates the refinement approach using box structure specification. A black box (BB_1) defines responses for a complete set of stimuli. BB_1 can be refined into a set of black boxes, $BB_{1.1}$ to $BB_{1.}n$, each of which addresses a class of behavior. Refinement continues until a cohesive class of behavior is identified (e.g., $BB_{1.1.1}$). A state box ($SB_{1.1.1}$) is then defined for the black box ($BB_{1.1.1}$). In this case, $SB_{1.1.1}$ contains all data and services required to implement the behavior defined by $BB_{1.1.1}$. Finally, $SB_{1.1.1}$ is refined into clear boxes ($CB_{1.1.1.n}$) and procedural design details are specified.

As each of these refinement steps occurs, verification of correctness also occurs. State-box specifications are verified to ensure that each conforms to the behavior defined by the parent black-box specification. Similarly, clear-box specifications are verified against the parent state box.

28.2.1 Black-Box Specification

A *black-box* specification describes an abstraction, stimuli, and response using the notation shown in Figure 28.3 [Mil88]. The function f is applied to a sequence S^* of inputs (stimuli) S and transforms them into an output (response) R. For simple software components, f may be a mathematical function, but in general, f is described using natural language (or a formal specification language).

FIGURE 28.3
A black-box specification



Many of the concepts introduced for object-oriented systems are also applicable for the black box. Data abstractions and the operations that manipulate those abstractions are encapsulated by the black box. Like a class hierarchy, the black-box specification can exhibit usage hierarchies in which low-level boxes inherit the properties of those boxes higher in the tree structure.

28.2.2 State-Box Specification

The *state box* is "a simple generalization of a state machine" [Mil88]. Recalling the discussion of behavioral modeling and state diagrams in Chapter 11, a state is some observable mode of system behavior. As processing occurs, a system responds to events (stimuli) by making a transition from the current state to some new state. As the transition is made, an action may occur. The state box uses a data abstraction to determine the transition to the next state and the action (response) that will occur as a consequence of the transition.

Referring to Figure 28.4, the state box incorporates a black box g. The stimulus S that is input to the black box arrives from some external source and a set of internal system states T. Mills [Mil88] provides a mathematical description of the function f of the black box contained within the state box:

$$g: S^* \times T^* \rightarrow R \times T$$

where g is a subfunction that is tied to a specific state t. When considered collectively, the state-subfunction pairs (t, g) define the black-box function f.

FIGURE 28.4
A state-box specification

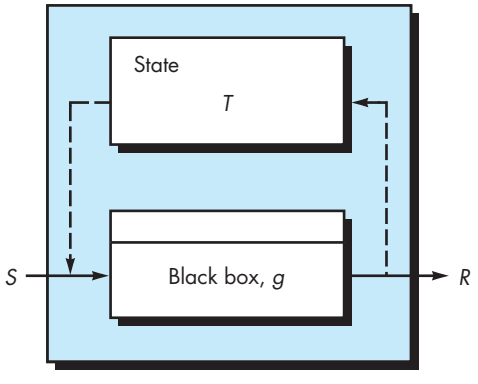
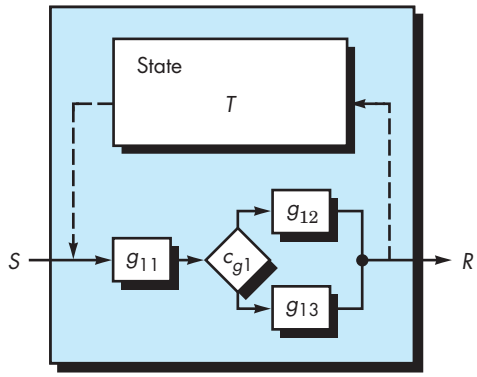


FIGURE 28.5

A clear-box specification



28.2.3 Clear-Box Specification

The clear-box specification is closely aligned with procedural design and structured programming. In essence, the subfunction *g* within the state box is replaced by the structured programming constructs that implement *g*.

As an example, consider the clear box shown in Figure 28.5. The black box *g*, shown in Figure 28.3, is replaced by a sequence construct that incorporates a conditional. These, in turn, can be refined into lower-level clear boxes as stepwise refinement proceeds.

It is important to note that the procedural specification described in the clear-box hierarchy can be proved to be correct. This topic is considered in Section 28.3.

28.3 CLEANROOM DESIGN

Cleanroom software engineering makes heavy use of the structured programming philosophy (Chapter 14). But in this case, structured programming is applied far more rigorously.

Basic processing functions (described during earlier refinements of the specification) are refined using a "stepwise expansion of mathematical functions into structures of logical connectives [e.g., if-then-else] and subfunctions, where the expansion [is] carried out until all identified subfunctions could be directly stated in the programming language used for implementation" [Dye92].

The structured programming approach can be used effectively to refine function, but what about data design? Here a number of fundamental design concepts (Chapter 12) come into play. Program data are encapsulated as a set of abstractions that are serviced by subfunctions. The concepts of data encapsulation, information hiding, and data typing are used to create the data design.

28.3.1 Design Refinement

Each clear-box specification represents the design of a procedure (subfunction) required to accomplish a state-box transition. Within the clear box, structured programming constructs and stepwise refinement are used to represent procedural detail. For example, a program function *f* is refined into a sequence of subfunctions *g* and *h*. These in turn are refined into conditional constructs (e.g., if-then-else and do-while). Further refinement continues until there is enough procedural detail to create the component in question.

KEY POINT

If you use structured programming constructs, a set of simple questions allows you to prove your code to be correct.

At each level of refinement, the cleanroom team[1] performs a *formal correctness verification*. To accomplish this, a set of generic correctness conditions are attached to the structured programming constructs. If a function *f* is expanded into a sequence *g* and *h*, the correctness condition for all input to *f* is

- Does *g* followed by *h* do *f*?

When a function *p* is refined into a conditional of the form, if <*c*> then *q*, else *r*, the correctness condition for all input to *p* is

- Whenever condition <*c*> is true, does *q* do *p*; and whenever <*c*> is false, does *r* do *p*?

When function *m* is refined as a loop, the correctness conditions for all input to *m* are

- Is termination guaranteed?
- Whenever <*c*> is true, does *n* followed by *m* do *m*; and whenever <*c*> is false, does skipping the loop still do *m*?

Each time a clear box is refined to the next level of detail, these correctness conditions are applied.

28.3.2 Design Verification

You should note that the use of the structured programming constructs constrains the number of correctness tests that must be conducted. A single condition is checked for sequences; two conditions are tested for if-then-else, and three conditions are verified for loops.

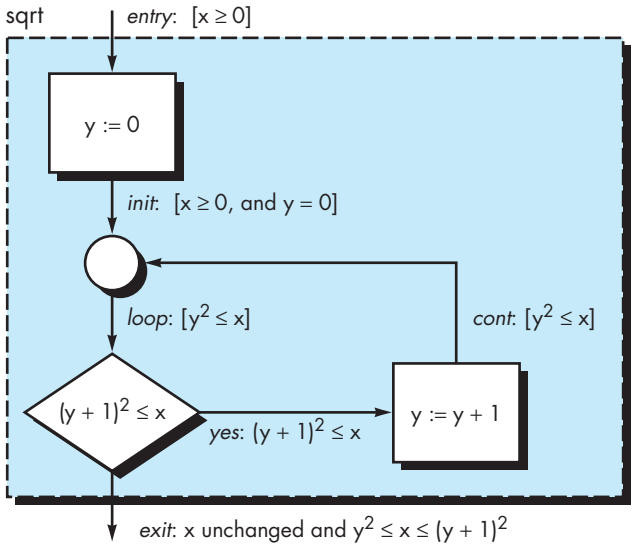
To illustrate correctness verification for a procedural design, we use a simple example first introduced by Linger, Mills, and Witt [Lin79]. The intent is to design and verify a small program that finds the integer part *y* of a square root of a given integer *x*. The procedural design is represented using the flowchart in Figure 28.6.[2]

1 Because the entire team is involved in the verification process, it is less likely that an error will be made in conducting the verification itself.

2 Figure 28.6 has been adapted from [Lin94]. Used with permission.

FIGURE 28.6

Computing the integer part of a square root
Source: [Lin79]



To verify the correctness of this design, entry and exit conditions are added as shown in Figure 28.6. The entry condition notes that x must be greater than or equal to 0. The exit condition requires that x remain unchanged and that y satisfy the expression noted in the figure. To prove the design to be correct, it is necessary to prove the conditions *init, loop, cont, yes,* and *exit* shown in Figure 28.6 are true in all cases. These are sometimes called *subproofs*.

1. The condition *init* demands that [$x \geq 0$ and $y = 0$]. Based on the requirements of the problem, the entry condition is assumed correct.[3] Therefore, the first part of the *init* condition, $x \geq 0$, is satisfied. Referring to the flowchart, the statement immediately preceding the *init* condition, sets $y = 0$. Therefore, the second part of the *init* condition is also satisfied. Hence, *init* is true.
2. The *loop* condition may be encountered in one of two ways: (1) directly from *init* (in this case, the *loop* condition is satisfied directly) or via control flow that passes through the condition *cont*. Since the *cont* condition is identical to the *loop* condition, *loop* is true regardless of the flow path that leads to it.
3. The *cont* condition is encountered only after the value of y is incremented by 1. In addition, the control flow path that leads to *cont* can be invoked only if the *yes* condition is also true. Hence, if $(y + 1)^2 \leq x$, it follows that $y^2 \leq x$. The *cont* condition is satisfied.

KEY POINT

To prove a design correct, you must first identify all conditions and then prove that each takes on an appropriate Bollean value. These are called *subproofs*.

3 A negative value for the square root has no meaning in this context.

4. The *yes* condition is tested in the conditional logic shown. Hence, the *yes* condition must be true when control flow moves along the path shown.
5. The *exit* condition first demands that x remain unchanged. An examination of the design indicates that x appears nowhere to the left of an assignment operator. There are no function calls that use x. Hence, it is unchanged. Since the conditional test $(y + 1)^2 \leq x$ must fail to reach the *exit* condition, it follows that $(y + 1)^2 \leq x$. In addition, the *loop* condition must still be true (i.e., $y^2 \leq x$). Therefore, $(y + 1)^2 > x$ and $y^2 \leq x$ can be combined to satisfy the exit condition.

You must further ensure that the loop terminates. An examination of the *loop* condition indicates that, because y is incremented and $x \geq 0$, the loop must eventually terminate.

The five steps just noted are a proof of the correctness of the design of the algorithm noted in Figure 28.6. You are now certain that the design will, in fact, compute the integer part of a square root.

A more rigorous mathematical approach to design verification is possible. However, a discussion of this topic is beyond the scope of this book. If you have interest, refer to [Lin79].

28.4 Cleanroom Testing

Quote:

"Quality is not an act, it is a habit."

Aristotle

The strategy and tactics of cleanroom testing are fundamentally different from conventional testing approaches (Chapters 22 through 26). Conventional methods derive a set of test cases to uncover design and coding errors. The goal of cleanroom testing is to validate software requirements by demonstrating that a statistical sample of use cases (Chapter 8) have been executed successfully.

28.4.1 Statistical Use Testing

ADVICE

Even if you decide against the cleanroom approach, it's worth considering statistical use testing as an integral part of your test strategy.

The user of a computer program rarely needs to understand the technical details of the design. The user-visible behavior of the program is driven by inputs and events that are often produced by the user. But in complex systems, the possible spectrum of input and events (i.e., the use cases) can be extremely broad. What subset of use cases will adequately verify the behavior of the program? This is the first question addressed by statistical use testing.

Statistical use testing "amounts to testing software the way users intend to use it" [Lin94]. To accomplish this, cleanroom testing teams (also called *certification teams*) must determine a usage probability distribution for the software. The specification (black box) for each increment of the software is analyzed to define a set of stimuli (inputs or events) that causes the software to change its behavior. Based on interviews with potential users, the creation of usage scenarios, and a general understanding of the application domain, a probability of use is assigned to each stimuli.

KEY POINT

Unlike conventional testing, cleanroom approaches are statistically driven.

Test cases are generated for each set of stimuli[4] according to the usage probability distribution. To illustrate, consider the *SafeHome* system discussed earlier in this book. Cleanroom software engineering is being used to develop a software increment that manages user interaction with the security system keypad. Five stimuli have been identified for this increment. Analysis indicates the percent probability distribution of each stimulus. To make selection of test cases easier, these probabilities are mapped into intervals numbered between 1 and 99 [Lin94] and illustrated in the following table:

Program Stimulus	Probability (%)	Interval
Arm/disarm (AD)	50	1–49
Zone set (ZS)	15	50–63
Query (Q)	15	64–78
Test (T)	15	79–94
Panic alarm	5	95–99

To generate a sequence of usage test cases that conform to the usage probability distribution, random numbers between 1 and 99 are generated. Each random number corresponds to an interval on the preceding probability distribution. Hence, the sequence of usage test cases is defined randomly but corresponds to the appropriate probability of stimuli occurrence. For example, assume the following random-number sequences are generated:

13-94-22-24-45-56

81-19-31-69-45-9

38-21-52-84-86-4

Selecting the appropriate stimuli based on the distribution interval shown in the table, the following use cases are derived:

AD–T–AD–AD–AD–ZS

T–AD–AD–AD–Q–AD–AD

AD–AD–ZS–T–T–AD

The testing team executes these use cases and verifies software behavior against the specification for the system. Timing for tests is recorded so that interval times may be determined. Using interval times, the certification team can compute mean-time-to-failure. If a long sequence of tests is conducted without failure, the MTTF is low and software reliability may be assumed high.

4 Automated tools may be used to accomplish this. For further information, see [Dye92].

28.4.2 Certification

The verification and testing techniques discussed earlier in this chapter lead to software components (and entire increments) that can be certified. Within the context of the cleanroom software engineering approach, *certification* implies that the reliability [measured by mean-time-to-failure (MTTF)] can be specified for each component.

The potential impact of certifiable software components goes far beyond a single cleanroom project. Reusable software components can be stored along with their usage scenarios, program stimuli, and probability distributions. Each component would have a certified reliability under the usage scenario and testing regime described. This information is invaluable to others who intend to use the components.

? How do we certify a software component?

The certification approach involves five steps [Woh94]: (1) usage scenarios must be created, (2) a usage profile is specified, (3) test cases are generated from the profile, (4) tests are executed and failure data are recorded and analyzed, and (5) reliability is computed and certified. Steps 1 through 4 have been discussed in a previous section. Certification for cleanroom software engineering requires the creation of three models [Poo93]:

> **Sampling model.** Software testing executes m random test cases and is certified if no failures or a specified number of failures occur. The value of m is derived mathematically to ensure that required reliability is achieved.
>
> **Component model.** A system composed of n components is to be certified. The component model enables the analyst to determine the probability that component i will fail prior to completion.
>
> **Certification model.** The overall reliability of the system is projected and certified.

At the completion of statistical use testing, the certification team has the information required to deliver software that has a certified MTTF computed using each of these models. If you have further interest, see [Cur86], [Mus87], or [Poo93] for additional detail.

28.5 Rethinking Formal Methods

Most software engineers agree that it is difficult if not impossible to create fault-free software systems following a modeling, design, code, and test paradigm. There are some systems that cannot be adequately tested before they are deployed (e.g., a robot operating on a remote planet under hostile environmental conditions).

Formal methods provide a method for verifying that a critical system will operate according to its specification. To do this software is treated as a mathematical

entity whose correctness can be proved using logical operations. It turns out that this is easier to do for programs implemented in imperative programming languages[5] than event-driven applications implemented in object-oriented languages.

Quote:

"Formal methods will never have a significant impact until they can be used by people that don't understand them."

Tom Melham

There are a number of potential benefits [Abr09] of using formal methods in trying to develop fault-free systems. Requirements written using natural languages are often ambiguous or incomplete. Formal methods model a system as a series of state transitions to represent what the developers of the system will observe as the program executes. The act of modeling the system may uncover several defects in the system. The modeling of a large software application requires several iterations.

Horizontal refinement elaborates software states from the abstract to the concrete by adding detail. This horizontal refinement is the basis for allowing traceability of software requirements. The assertions used to prohibit the software from reaching an invalid state can be defined and their placement verified. Once horizontal refinement of this discrete model is complete, *vertical refinement* is used to transform the states and transitions so that they can be implemented in the target programming language. Vertical refinement is the process that attempts to "glue" the abstract to the concrete, without allowing communication weaknesses in a poorly chosen target language to affect the requirement specifications.

The integration of legacy code complicates the use of formal methods since the legacy requirements may not be a good fit for the new system. In many instances, it is best to capture the legacy code behavior in the specification, and then implement the behavior in new code.

Some opponents of formal methods argue that many of its practices are contrary to the tenets of agile process models. However, Black and her colleagues [Bla09] suggest that elements of both formal methods and agile processes can be combined to create better software products. Both have the same basic goal of trying to create reliable software. Formal methods can add value to agile development by forcing developers to make sure the system safety property axioms[6] are valid.

Formal methods techniques (e.g., static analysis and theorem-proving tools) can be used to automatically generate test cases from system models and indicate where assertions should be placed in the evolving program code. Informal requirements can be translated into formal notation that is embedded into the source code. Assertions written in formal notation may be machine-checked for inconsistencies. Code often needs to be refactored[7] as it evolves. Formal methods can provide the basis for defining correctness-preserving transformations to ensure the refactored code still meets it requirements.

5 An *imperative language programming language* achieves its primary affect by assigning the values of algebraic expressions.

6 Safety property axioms are statements about what software may not allow. Safety was discussed as part of security in Chapter 27.

7 *Refactoring* (Chapter 5) improves the code without changing its meaning.

Although the use of formal methods may slow the delivery of the first software increment, it can reduce the amount of rework during the project lifetime and therefore provide a significant return on the time investment applied. If the customer's requirements are extremely volatile, there is no guarantee that a project can be completed more quickly using agile techniques. The name of the game is creating a product that that meets the customer's requirements, and supplementing an agile approach with formal methods can help ensure that requirements have been met.

Mayer and his colleagues [Mey09] describe three types of tools that can be useful in automating testing, thereby acting to facilitate statistical use testing approaches. *Test generation* tools create and execute test cases without human input. *Test extraction* tools produce test cases for later replay from program execution failures. *Manual test integration* tools provide help in developing and managing test cases that are produced manually.

The part of testing that is most likely to be automated is test case execution, which is very important for efficient regression testing. Meyer [Mey09] suggests that test case generation is relatively straightforward for languages such as Eiffel that support design by contract.[8] Test cases are generated to exercise preconditions and postconditions represented as part of the source code (discussed in Section 28.6). When developers find bugs in programs, they typically log them and fix them, missing the opportunity to create test cases for future regression testing. When an error occurs during program execution, testing tools capture the information about precondition violations or postcondition failures and convert them into test cases automatically.

SOFTWARE TOOLS

Formal Methods

Objective: The objective of formal methods tools is to assist a software team in specification and correctness verification.

Mechanics: Tools mechanics vary. In general, tools assist in specification and automating correctness proving, usually by defining a specialized language for theorem proving. Many tools are not commercialized and have been developed for research purposes.

Representative Tools:[9]

ACL2, developed at the University of Texas (**www.cs.utexas.edu/users/moore/acl2/**), is "both a programming language in which you can model computer systems and a tool to help you prove properties of those models."

Pointers to several formal methods tool repositories can be found on a site hosted by Cyber Security and Information Systems Information Analysis Center (CSIAC) **https://sw.thecsiac.com/databases/url/key/53/57#.UHrvKYbwpuh/**.

8 *Design by contract* uses preconditions, postconditions, and invariants to answer the questions: What does the program expect? What does it guarantee? What does it maintain?

9 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

28.6 Formal Methods Concepts

The Encyclopedia of Software Engineering [Mar01] defines formal methods in the following manner:

> Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.

KEY POINT

A formal specification should have consistency, completeness, and lack of ambiguity.

The desired properties of a formal specification are the objectives of all specification methods. However, the mathematically based specification language used for formal methods results in a much higher likelihood of achieving these properties. The formal syntax of a specification language (Appendix 3) enables requirements or design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or a graphical notation (e.g., UML) must be interpreted by a reader. The descriptive facilities of set theory and logic notation enable a clear statement of requirements. To be consistent, requirements stated in one place in a specification should not be contradicted in another place. Consistency is achieved[10] by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification.

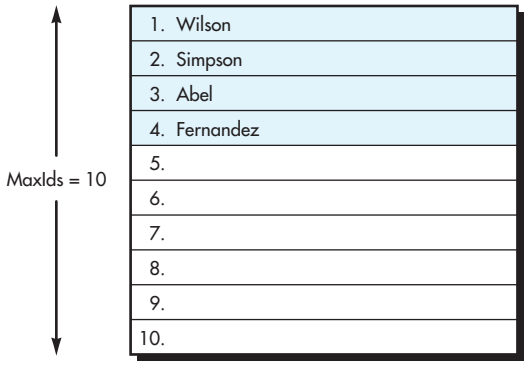
To introduce basic formal methods concepts, let's consider a few simple examples to illustrate the use of mathematical specification, without getting bogged down in too much mathematical detail.

Example 1: A symbol table. A program is used to maintain a symbol table. Such a table is used frequently in many different types of applications. It consists of a collection of items without any duplication. An example of a typical symbol table is shown in Figure 28.7. It represents the table used by an operating system to hold the names of the users of the system. Other examples of tables include the collection of names of staff in a payroll system, the collection of names of computers in a network communications system, and the collection of destinations in a system for producing transportation timetables.

Assume that the table presented in this example consists of no more than *MaxIds* names. This statement, which places a constraint on the table, is a component of a condition known as a *data invariant*. A data invariant is a condition that is true throughout the execution of the system that contains a collection

10 In reality, completeness is difficult to ensure, even when formal methods are used. Some aspects of a system may be left undefined as the specification is being created; other characteristics may be purposely omitted to allow designers some freedom in choosing an implementation approach; and finally, it is impossible to consider every operational scenario in a large, complex system. Things may simply be omitted by mistake.

FIGURE 28.7
A symbol table



of data. The data invariant that holds for the symbol table just discussed has two components: (1) that the table will contain no more than *MaxIds* names and (2) that there will be no duplicate names in the table. In the case of the symbol table program, this means that no matter when the symbol table is examined during execution of the system, it will always contain no more than *MaxIds* names and will contain no duplicates.

Another important concept is that of a *state*. Many formal languages use the notion of states as they were discussed in Chapter 11; that is, a system can be in one of several states, each representing an externally observable mode of behavior. However, a different definition for the term *state* is used in some specification languages. In these languages the state of a system is represented by the system's stored data. Using the latter definition in the example of the symbol table program, the state is the symbol table.

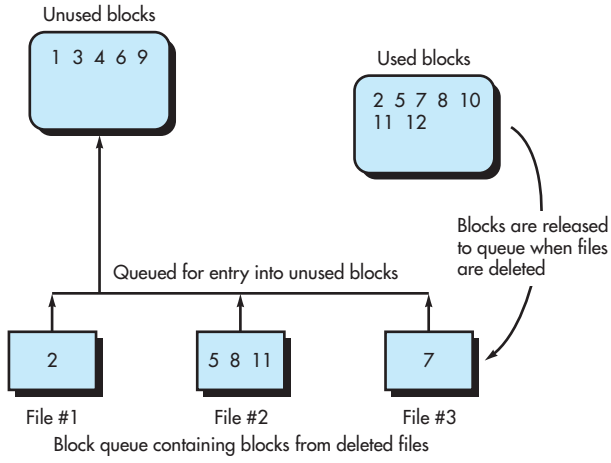
The final concept is that of an *operation*. This is an action that takes place within a system and reads or writes data. If the symbol table program is concerned with adding and removing names from the symbol table, then it will be associated with two operations: an operation to *add* a specified name to the symbol table and an operation to *remove()* an existing name from the table.[11] If the program provides the facility to check whether a specific name is contained in the table, then there would be an operation that would return some indication of whether the name is in the table.

Three types of conditions can be associated with operations: invariants, preconditions, and postconditions. An *invariant* defines what is guaranteed not to change. For example, the symbol table has an invariant that states that the number of elements is always less than or equal to *MaxIds*. A *precondition* defines the circumstances in which a particular operation is valid. For example, the precondition for an operation that adds a name to a staff identifier symbol table is valid

11 It should be noted that adding a name cannot occur in the *full* state and deleting a name is impossible in the *empty* state.

FIGURE 28.8

A block handler



only if the name that is to be added is not contained in the table and also if there are fewer than *MaxIds* staff identifiers in the table. The *postcondition* of an operation defines what is guaranteed to be true upon completion of an operation. This is defined by its effect on the data. For the *add()* operation, the postcondition would specify mathematically that the table has been augmented with the new identifier.

Example 2: A block handler. One of the more important parts of a simple operating system is the subsystem that maintains files created by users. Part of the filing subsystem is the *block handler*. Files in the file store are composed of blocks of storage that are held on a file storage device. During the operation of the computer, files will be created and deleted, requiring the acquisition and release of blocks of storage. In order to cope with this, the filing subsystem will maintain a reservoir of unused (free) blocks and keep track of blocks that are currently in use. When blocks are released from a deleted file, they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This is shown in Figure 28.8. In this figure, a number of components are shown: the reservoir of unused blocks, the blocks that currently make up the files administered by the operating system, and those blocks that are waiting to be added to the reservoir. The waiting blocks are held in a queue, with each element of the queue containing a set of blocks from a deleted file.

For this subsystem the state is the collection of free blocks, the collection of used blocks, and the queue of returned blocks. The data invariant, expressed in natural language, is

- No block will be marked as both unused and used.
- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.

- No elements of the queue will contain the same block numbers.
- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.
- The collection of unused blocks will have no duplicate block numbers.
- The collection of used blocks will have no duplicate block numbers.

Some of the operations associated with this data are: *add()* a collection of blocks to the end of the queue, *remove()* a collection of used blocks from the front of the queue and place them in the collection of unused blocks, and *check()* whether the queue of blocks is empty.

The precondition of *add()* is that the blocks to be added must be in the collection of used blocks. The postcondition is that the collection of blocks is now found at the end of the queue. The precondition of *remove()* is that the queue must have at least one item in it. The postcondition is that the blocks must be added to the collection of unused blocks. The *check()* operation has no precondition. This means that the operation is always defined, regardless of what value the state is. The postcondition delivers the value *true* if the queue is empty and *false* otherwise.

In the examples noted in this section, we introduce the key concepts of formal specification but without emphasizing the mathematics that are required to make the specification formal. In Appendix 3 we consider how mathematical notation can be used to formally specify some element of a system.

28.7 Alternative Arguments

Parnas [Par10] offers several criticisms of the approach typically used to prove program correctness. He argues that using values of expression variables to define the state of computation ignores the role of interrupt handlers in real-time systems. Formal methods typically discourage the use of expression side effects that may be common in some application domains. Information hiding facilitated by use of abstract data types gives rise to the use of hidden states, which must be accounted for when writing data invariants. This often makes assertions more complex than the code itself. Formal methods were devised for deterministic programs.[12] This does not account for programs designed to run forever.

Formal methods depend on the use of models intended to provide a simplified view of the program. There is no reliable method of translating these models to program code. It is possible to prove a program correct that may have defects, because the model or code was flawed.

12 *Deterministic programs* always begin in a defined start state and end in a single end state.

In formal methods work, the process of proving software correctness becomes the task of showing that a program can be reduced to a sequence of legal states[13] or data transitions. This may not be sufficient for all programs. Predicates that can verify correct run-time behavior of a program may be needed to check the correctness of real-time or parallel algorithms.

Assertions help developers document and understand software more effectively but are only practical for small programs. The use of assertions does not eliminate the need for external documentation which is often imprecise due to ambiguities in the natural language used to describe the system. In addition, formal methods do not help software engineers select from among several correct software designs.

28.8 Summary

Cleanroom software engineering is a formal approach to software development that can lead to software that has remarkably high quality. It uses box structure specification for analysis and design modeling and emphasizes correctness verification, rather than testing, as the primary mechanism for finding and removing errors. Statistical use testing is applied to develop the failure rate information necessary to certify the reliability of delivered software.

The cleanroom approach begins with analysis and design models that use a box structure representation. A "box" encapsulates the system (or some aspect of the system) at a specific level of abstraction. Black boxes are used to represent the externally observable behavior of a system. State boxes encapsulate state data and operations. A clear box is used to model the procedural design that is implied by the data and operations of a state box.

Correctness verification is applied once the box structure design is complete. The procedural design for a software component is partitioned into a series of subfunctions. To prove the correctness of the subfunctions, exit conditions are defined for each subfunction and a set of subproofs is applied. If each exit condition is satisfied, the design must be correct.

Once correctness verification is complete, statistical use testing commences. Unlike conventional testing, cleanroom software engineering does not emphasize unit or integration testing. Rather, the software is tested by defining a set of usage scenarios, determining the probability of use for each scenario, and then defining random tests that conform to the probabilities. The error records that result are combined with sampling, component, and certification models to enable mathematical computation of projected reliability for the software component.

13 A legal state is a set of program variables that were assigned values allowed by the processing the program inputs.

Formal methods use the descriptive facilities of set theory and logic notation to enable a software engineer to create a clear statement of facts (requirements). The underlying concepts that govern formal methods are: (1) the data invariant, a condition true throughout the execution of the system that contains a collection of data; (2) the state, a representation of a system's externally observable mode of behavior, or (in Z and related languages) the stored data that a system accesses and alters; and (3) the operation, an action that takes place in a system and reads or writes data to a state. An operation is associated with two conditions: a precondition and a postcondition.

Will cleanroom software engineering or formal methods ever be widely used? The answer is probably not. They are more difficult to learn than conventional software engineering methods and represent significant "culture shock" for some software practitioners. But the next time you hear someone lament, "Why can't we get this software right the first time?" you'll know that there are techniques that can help you to do exactly that.

Problems and Points to Ponder

28.1. If you had to pick one aspect of cleanroom software engineering that makes it radically different from conventional or object-oriented software engineering approaches, what would it be?

28.2. How do an incremental process model and certification work together to produce high-quality software?

28.3. Using box structure specification, develop "first-pass" analysis and design models for the *SafeHome* system.

28.4. A bubble-sort algorithm is defined in the following manner:

```
procedure bubblesort;
var i, t, integer;
begin
repeat until t=a[1]
        t:=a[1];
        for j:= 2 to n do
            if a[j-1] > a[j] then begin
                t:=a[j-1];
                a[j-1]:=a[j];
                a[j]:=t;
                end
endrep
end
```

Partition the design into subfunctions, and define a set of conditions that would enable you to prove that this algorithm is correct.

28.5. Document a correctness verification proof for the bubble sort discussed in Problem 28.4.

28.6. Select a program that you use regularly (e.g., an e-mail handler, a word processor, a spreadsheet program). Create a set of usage scenarios for the program. Define the probability of use for each scenario, and then develop a program stimuli and probability distribution table similar to the one shown in Section 28.4.1.

28.7. For the program stimuli and probability distribution table developed in Problem 28.6, use a random-number generator to develop a set of test cases for use in statistical use testing.

28.8. In your own words, describe the intent of certification in the cleanroom software engineering context.

28.9. You have been assigned to a team that is developing software for a fax modem. Your job is to develop the "phone book" portion of the application. The phone book function enables up to *MaxNames* people to be stored along with associated company names, fax numbers, and other related information. Using natural language, define

a. The data invariant.
b. The state.
c. The operations that are likely.

28.10. You have been assigned to a software team that is developing software called MemoryDoubler that provides greater apparent memory for a PC than physical memory. This is accomplished by identifying, collecting, and reassigning blocks of memory that have been assigned to an existing application but are not being used. The unused blocks are reassigned to applications that require additional memory. Making appropriate assumptions and using natural language, define

a. The data invariant.
b. The state.
c. The operations that are likely.

Further Readings and Information Sources

Relatively few books on advanced specification and verification techniques have been published in recent years. However, some new additions to the literature are worth considering. Books by Gabbar (*Modern Formal Methods and Applications,* Springer, 2010) and Boca and his colleagues (*Formal Methods: State of the Art and New Directions,* Springer, 2010) present both fundamentals, new developments, and advanced applications. Jackson (*Software Abstractions,* 2nd ed., MIT Press, 2012) presents all of the basics and an approach that he calls "lightweight formal methods." Monin and Hinchey (*Understanding Formal Methods,* Springer, 2003) provide an excellent introduction to the subject. Butler and other editors (*Integrated Formal Methods,* Springer, 2002) present a variety of papers on formal methods topics.

In addition to books referenced in this chapter, Prowell and his colleagues (*Cleanroom Software Engineering: Technology and Process,* Addison-Wesley, 1999) provide an in-depth treatment of all important aspects of the cleanroom approach. Useful discussions of cleanroom topics have been edited by Liu (*Formal Engineering for Industrial Software Development: Using the SOFL Method,* Springer, 2010) and Poore and Trammell (*Cleanroom Software Engineering: A Reader,* Blackwell Publishing, 1996). Becker and Whittaker (*Cleanroom Software Engineering Practices,* Idea Group Publishing, 1997) present an excellent overview for those who are unfamiliar with cleanroom practices.

The Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, April 1995) contains reprints of a number of important articles. The Cyber Security and Information Systems Information Analysis Center (CSIAC) (**www.thecsiac.com**) provides many useful papers, guidebooks, and other information sources on cleanroom software engineering.

Design verification via proof of correctness lies at the heart of the cleanroom approach. Books by Cupillari (*The Nuts and Bolts of Proofs,* 4th ed., Academic Press, 2012), Solow (*How to Read and Do Proofs,* 5th ed., Wiley, 2009), Eccles (*An Introduction to Mathematical Reasoning,* Cambridge University Press, 1998), provide excellent introductions into the mathematical basics. Stavely (*Toward Zero-Defect Software,* Addison-Wesley, 1998), Baber (*Error-Free*

Software, Wiley, 1991), and Schulmeyer (*Zero Defect Software,* McGraw-Hill, 1990) discuss proof of correctness in considerable detail.

In the formal methods domain, books by Hinchey and Bowan (*Industrial Strength Formal Methods,* Springer-Verlag, 1999) and Hussmann (*Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997). Appendix 3 of this text contains additional discussion on the mathematical foundations of formal methods and the role of specification languages in software engineering.

A wide variety of information sources on cleanroom software engineering and formal methods is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

29

Software Configuration Management

Key Concepts

baselines 626
change control.... 635
configuration audit 639
configuration management, elements of..... 626
configuration objects..... 642
content management..... 643
identification..... 633
impact management..... 638
repository 630
SCM process..... 632

Change is inevitable when computer software is built and can lead to confusion when you and other members of a software team are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error. Babich [Bab86] discusses this when he states:

> The art of coordinating software development to minimize . . . confusion is called configuration management. Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change,

Quick Look

What is it? When you build computer software, change happens. And because it happens, you need to manage it effectively. Software configuration management (SCM), also called change management, is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Who does it? Everyone involved in the software process is involved with change management to some extent, but specialized support positions are sometimes created to manage the SCM process.

Why is it important? If you don't control change, it controls you. And that's never good. It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. As a consequence, software quality suffers and delivery is delayed. For that reason, change management is an essential part of quality management.

What are the steps? Because many work products are produced when software is built, each must be uniquely identified. Once this is accomplished, mechanisms for version and change control can be established. To ensure that quality is maintained as changes are made, the process is audited; and to ensure that those with a need to know are informed about changes, reporting is conducted.

What is the work product? A Software Configuration Management Plan defines the project strategy for change management. In addition, when formal SCM is invoked, the change control process produces software change requests, reports, and engineering change orders.

How do I ensure that I've done it right? When every work product can be accounted for, traced, and controlled; when every change can be tracked and analyzed; when everyone who needs to know about a change has been informed—you've done it right.

software configuration items 628
status reporting . . 639
version control . . . 634
Web and MobileApps 640

(3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that are initiated when a software engineering project begins and terminates only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter, we discuss the specific activities that enable you to manage change.

29.1 Software Configuration Management

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms), (2) work products that describe the computer programs (targeted at various stakeholders), and (3) data or content (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration*.

Quote:

"There is nothing permanent except change."

Heraclitus, 500 B.C.

As software engineering work progresses, a hierarchy of *software configuration items* (SCIs)—a named element of information that can be as small as a single UML diagram or as large as the complete design document—is created. If each SCI simply led to other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change*. Change may occur at any time, for any reason. In fact, the *First Law of System Engineering* [Ber80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

? What is the origin of changes that are requested for software?

- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we describe major SCM tasks and important concepts that help us to manage change.

29.1.1 An SCM Scenario[1]

A typical configuration management (CM) operational scenario involves a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product. In the scenario, assume that the product is a small one involving about 15,000 lines of code being developed by a team of four people. (Note that other scenarios of smaller or larger teams are possible, but, in essence, there are generic issues that each of these projects face concerning CM.)

? What are the goals of and the activities performed by each of the constituencies involved in change management?

At the operational level, the scenario involves various roles and tasks. For the project manager, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

The goals of the configuration manager are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating them (via a Change Control Board that is responsible for approving changes to the software system), and for authorizing changes. The manager creates and disseminates task lists for the engineers and basically creates the project context. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

KEY POINT

There must be a mechanism to ensure that simultaneous changes to the same component are properly tracked, managed and executed.

For the software engineers, the goal is to work effectively. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting work products. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying one another about tasks required and tasks completed. Changes are propagated across each other's work by merging files. Mechanisms exist to ensure that, for components that undergo simultaneous changes, there is

1 This section is extracted from [Dar01]. Special permission to reproduce "Spectrum of Functionality in CM System" by Susan Dart [Dar01], © 2001 by Carnegie Mellon University is granted by the Software Engineering Institute.

some way of resolving conflicts and merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed. The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

The customer uses the product. Since the product is under CM control, the customer follows formal procedures for requesting changes and for indicating bugs in the product.

Ideally, a CM system used in this scenario should support all these roles and tasks; that is, the roles determine the functionality required of a CM system. The project manager sees CM as an auditing mechanism; the configuration manager sees it as a controlling, tracking, and policy-making mechanism; the software engineer sees it as a changing, building, and access control mechanism; and the customer sees it as a quality assurance mechanism.

29.1.2 Elements of a Configuration Management System

In her comprehensive white paper on software configuration management, Susan Dart [Dar01] identifies four important elements that should exist when a configuration management system is developed:

- *Component elements*—A set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—A collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- *Construction elements*—A set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—A set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

These elements (to be discussed in more detail in later sections) are not mutually exclusive. For example, component elements work in conjunction with construction elements as the software process evolves. Process elements guide many human activities that are related to SCM and might therefore be considered human elements as well.

29.1.3 Baselines

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want

ADVICE

Most software changes are justified, so there's no point in complaining about them. Rather, be certain that you have mechanisms in place to handle them.

to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, and how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: *Most changes are justified!*

A *baseline* is a software configuration management concept that helps you to control change without seriously impeding justifiable change. The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

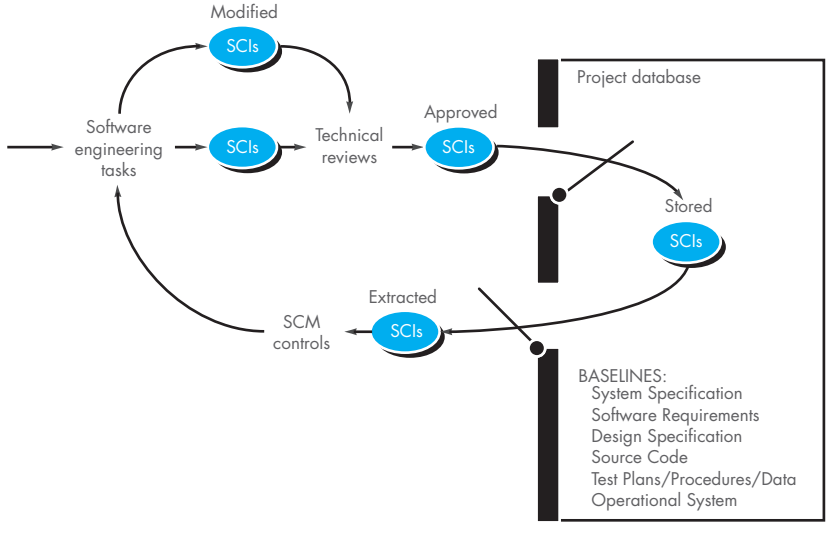
> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review (Chapter 20). For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 29.1.

FIGURE 29.1

Baselined SCIs and the project database



ADVICE

Be sure that the project database is maintained in a centralized, controlled location.

The progression of events that lead to a baseline is also illustrated in Figure 29.1. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository* and discussed in Section 29.3). When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private workspace. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 29.1 illustrate the modification path for a baselined SCI.

29.1.4 Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, an SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, or a named program component).

In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, browsers, and other automated tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A *configuration object* has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 29.2, the configuration objects, **DesignSpecification, DataModel, ComponentN, SourceCode,** and **TestSpecification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **DataModel** and **ComponentN** are part of the object **DesignSpecification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **SourceCode** object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.[2]

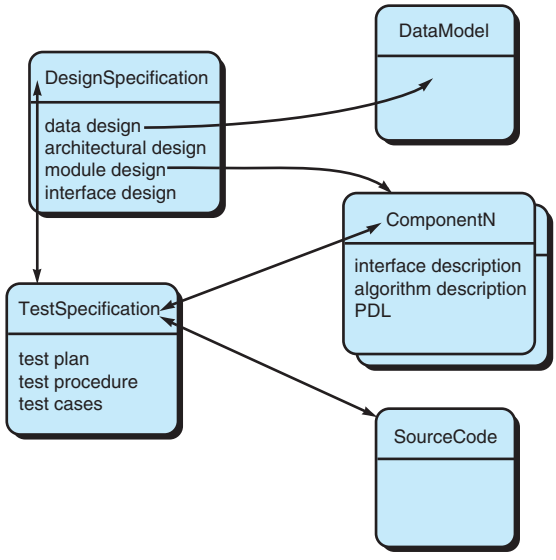
29.1.5 Management of Dependencies and Changes

We introduced the concept of traceability and the use of traceability matrices in Section 8.2.6. The traceability matrix is one way to document dependencies

2 These relationships are defined within the database. The structure of the database (repository) is discussed in greater detail in Section 29.3.

FIGURE 29.2

Configuration objects



among requirements, architectural decisions (Section 13.5), and defect causes (Section 21.6). These dependencies need to be taken into account when determining the impact of a proposed change and guiding the selection of test cases that should be used for regression testing (Section 22.3.2). Dependency management can be viewed as impact management.[3] This helps developers to focus on how changes that are made affect their work [Sou08].

Impact analysis focuses on organizational behavior as well as individual actions. Impact management involves two complementary aspects: (1) ensuring that software developers employ strategies to minimize the impact of their colleagues' actions on their own work, and (2) encouraging software developers to use practices that minimize the impact of their own work on that of their colleagues. It is important to note that when a developer tries to minimize the impact of her work on others she is also reducing the work others need to do to minimize the impact of her work on theirs [Sou08].

It is important to maintain software work products to ensure that developers are aware of the dependencies among the SCIs. Developers must establish discipline when checking items in and out of the SCM repository and when making approved changes as discussed in Section 29.2. Bug tracking software is also useful in helping to uncover SCI dependencies. Electronic communication (e-mail, wikis, social networks) provide convenient ways for developers to share undocumented dependencies and problems as they arise.

3 Impact management is discussed further in Section 29.3.4

29.2 The SCM Repository

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.

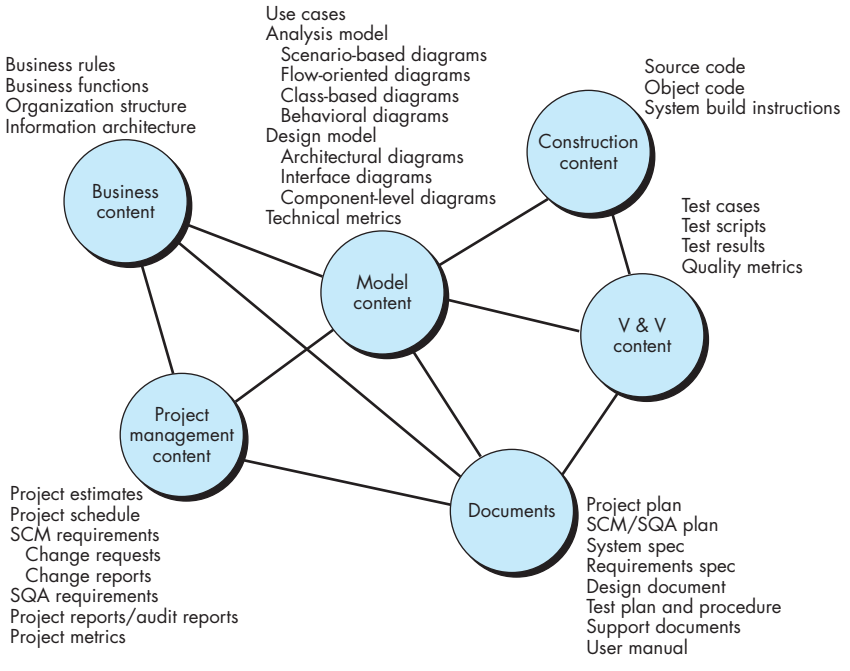
To achieve these capabilities, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs.

29.2.1 General Features and Content

The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. A detailed breakdown of types of representations, documents, and other work products that are stored in the repository is presented in Figure 29.3.

Figure 29.3

Content of the repository



WebRef

An example of a commercially available repository can be obtained at **www.oracle.com/technology/products/repository/index.html.**

A robust repository provides two different classes of services: (1) the same types of services that might be expected from any sophisticated database management system and (2) services that are specific to the software engineering environment.

A repository that serves a software engineering team should also (1) integrate with or directly support process management functions, (2) support specific rules that govern the SCM function and the data maintained within the repository, (3) provide an interface to other software engineering tools, and (4) accommodate storage of sophisticated data objects (e.g., text, graphics, video, audio).

29.2.2 SCM Features

KEY POINT

The repository must be capable of maintaining SCIs related to many different versions of the software. More important, it must provide the mechanisms for assembling these SCIs into a version-specific configuration.

To support SCM, the repository must have a tool set that provides support for the following features:

Versioning. As a project progresses, many versions (Section 29.3.2) of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.

The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects such as screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

Dependency tracking and change management. The repository manages a wide variety of relationships among the data elements stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on. Some of these relationships are merely associations, and some are dependencies or mandatory relationships.

The ability to keep track of all of these relationships is crucial to the integrity of the information stored in the repository and to the generation of deliverables based on it, and it is one of the most important contributions of the repository concept to the improvement of the software development process. For example, if a UML class diagram is modified, the repository can detect whether related classes, interface descriptions, and code components also require modification and can bring affected SCIs to the developer's attention.

Requirements tracing. This special function depends on link management and provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, it provides the ability to identify which requirement generated any given work product (backward tracing).

Configuration management. A configuration management facility keeps track of a series of configurations representing specific project milestones or production releases.

Audit trails. An audit trail establishes additional information about when, why, and by whom changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository. A repository trigger mechanism is helpful for prompting the developer or the tool that is being used to initiate entry of audit information (such as the reason for a change) whenever a design element is modified.

29.3 The SCM Process

Quote:

"Any change, even a change for the better, is accompanied by drawbacks and discomforts."

Arnold Bennett

The software configuration management process defines a series of tasks that have four primary objectives: (1) to identify all items that collectively define the software configuration, (2) to manage changes to one or more of these items, (3) to facilitate the construction of different versions of an application, and (4) to ensure that software quality is maintained as the configuration evolves over time.

A process that achieves these objectives need not be bureaucratic and ponderous, but it must be characterized in a manner that enables a software team to develop answers to a set of complex questions:

? What questions should the SCM process be designed to answer?

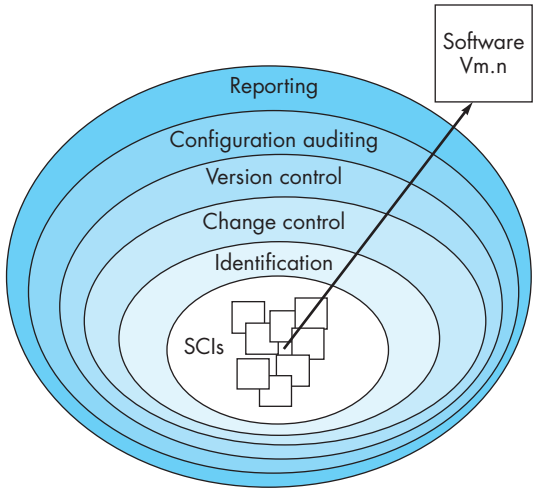
- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- How does an organization assess the impact of change and manage the impact effectively?
- Who has responsibility for approving and ranking requested changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead to the definition of five SCM tasks—identification, version control, change control, configuration auditing, and reporting—illustrated in Figure 29.4.

Referring to the figure, SCM tasks can be viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system. As an SCI moves through a layer, the actions implied by each SCM

FIGURE 29.4

Layers of the SCM process



task may or may not be applicable. For example, when a new SCI is created, it must be identified. However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play). A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know. In the sections that follow, we examine each of these SCM process layers in more detail.

29.3.1 Identification of Objects in the Software Configuration

To control and manage software configuration items, each should be separately named and then organized using an object-oriented approach. Two types of objects can be identified [Cho89]: basic objects and aggregate objects.[4] A *basic object* is a unit of information that you create during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, part of a design model, source code for a component, or a suite of test cases that are used to exercise the code. An *aggregate object* is a collection of basic objects and other aggregate objects. For example, a **DesignSpecification** is an aggregate object. Conceptually, it can be viewed as a named (identified) list of pointers that specify aggregate objects such as **ArchitecturalModel** and **DataModel,** and *basic objects* such as **ComponentN** and **UMLDiagramN**.

KEY POINT

The interrelationships established for configuration objects allow you to assess the impact of change.

Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously. The object description is a list of

4 The concept of an aggregate object [Gus89] has been proposed as a mechanism for representing a complete version of a software configuration.

data items that identify the SCI type (e.g., model element, program, data) represented by the object, a project identifier, and change and/or version information. Resources are "entities that are provided, processed, referenced or otherwise required by the object" [Cho89]. For example, data types, specific functions, or even variable names may be considered to be object resources. The realization is a pointer to the "unit of text" for a basic object and null for an aggregate object.

Configuration object identification can also consider the relationships that exist between named objects. For example, using the simple notation

Class diagram <part-of> requirements model;
Requirements model <part-of> requirements specification;

we create a hierarchy of SCIs.

ADVICE

Even if the project database provides the ability to establish these relationships, they are time-consuming to establish and difficult to keep up to date. Although very useful for impact analysis, they are not essential for overall change management.

In many cases, objects are interrelated across branches of the object hierarchy. These cross-structural relationships can be represented in the following manner:

data model <interrelated> data flow model;
data model <interrelated> test case class m;

In the first case, the interrelationship is between a composite object, while the second relationship is between an aggregate object (**DataModel**) and a basic object (**TestCaseClassM**).

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent.

29.3.2 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities: (1) a project database (repository) that stores all relevant configuration objects, (2) a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions), (3) a *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software. In addition, version control and change control systems often implement an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

A number of version control systems establish a *change set*—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software. Dart [Dar91] notes that a change set "captures all

changes to all files in the configuration along with the reason for changes and details of who made the changes and when."

A number of named change sets can be identified for an application or system. This enables you to construct a version of the software by specifying the change sets (by name) that must be applied to the baseline configuration. To accomplish this, a *system modeling* approach is applied. The system model contains: (1) a *template* that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed, (2) construction rules, and (3) verification rules.[5]

A number of different automated approaches to version control have been proposed over the years. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

SOFTWARE TOOLS

The Concurrent Versions System (CVS)

The use of tools to achieve version control is essential for effective change management. The *Concurrent Versions System* (CVS) is a widely used tool for version control. Originally designed for source code but useful for any text-based file, the CVS system (1) establishes a simple repository, (2) maintains all versions of a file in a single named file by storing only the differences between progressive versions of the original file, and (3) protects against simultaneous changes to a file by establishing different directories for each developer, thus insulating one from another. CVS merges changes when each developer completes her work.

It is important to note that CVS is not a "build" system; that is, it does not construct a specific version of the software. Other tools (e.g., *Makefile*) must be integrated with CVS to accomplish this. CVS does not implement a change control process (e.g., change requests, change reports, bug tracking).

Even with these limitations, CVS "is a dominant open-source network-transparent version control system [that] is useful for everyone from individual developers to large, distributed teams" [CVS07]. Its client-server architecture allows users to access files via Internet connections, and its open-source philosophy makes it available on most popular platforms.

CVS is available at no cost for Windows, Mac OS, LINUX, and UNIX environments and an open-source version of the application [CVS12] is available.[6]

Quote:

"The art of progress is to preserve order amid change and to preserve change amid order."

Alfred North Whitehead

29.3.3 Change Control

The reality of change control in a software engineering context has been summed up beautifully by James Bach [Bac98]:

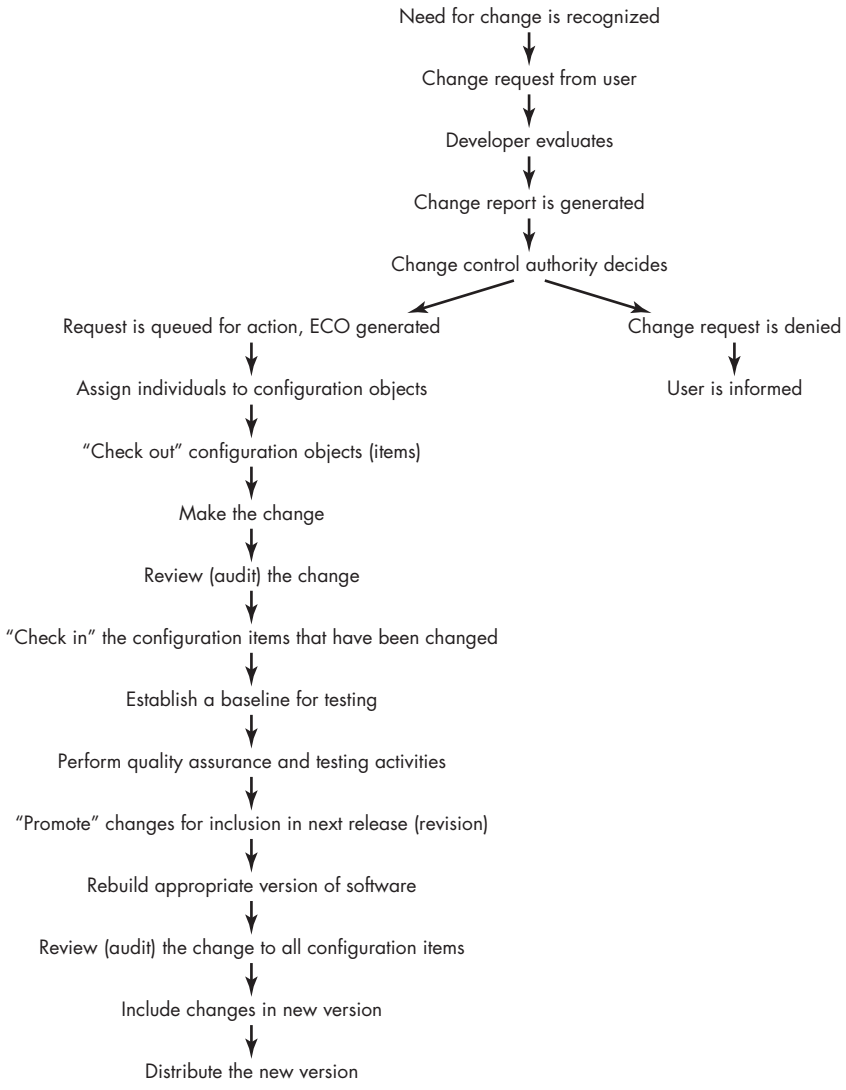
> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

5 It is also possible to query the system model to assess how a change in one component will impact other components.

6 Download link for [CVS12] **http://olex.openlogic.com/packages/cvs**

FIGURE 29.5

The change control process



KEY POINT

It should be noted that a number of change requests may be combined to result in a single ECO and that ECOs typically result in changes to multiple configuration objects.

Bach recognizes that we face a balancing act. Too much change control and we create problems. Too little, and we create other problems.

For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 29.5. A *change request* is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group that makes a final decision

on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit.

The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made. As an alternative, the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied. The object(s) is (are) then "checked in" to the database and appropriate version control mechanisms (Section 29.3.2) are used to create the next version of the software.

These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

You may feel uncomfortable with the level of bureaucracy implied by the change control process description shown in Figure 29.5. This feeling is not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately, many have none) have created a number of layers of control to help avoid the problems alluded to here.

ADVICE

Opt for a bit more change control than you think you'll need. It's likely that too much will be the right amount.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone technical review and has been approved, a baseline can be created.[7] Once an SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, the developer dispenses with the formal generation of change requests, change reports, and ECOs. However, assessment of each change is conducted and all changes are tracked and reviewed.

Quote:

"Change is inevitable, except for vending machines."

Bumper sticker

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 29.5.

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the

7 A baseline can be created for other reasons as well. For example, when "daily builds" are created, all components checked in by a given time become the baseline for the next day's work.

CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify customers' perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

SafeHome

SCM Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I know it's early, but we've got to talk about change management.

Vinod (laughing): Hardly. Marketing called this morning with a few "second thoughts." Nothing major, but it's just the beginning.

Jamie: We've been pretty informal about change management on past projects.

Doug: I know, but this is bigger and more visible, and as I recall . . .

Vinod (nodding): We got killed by uncontrolled changes on the home lighting control project . . . remember the delays that . . .

Doug (frowning): A nightmare that I'd prefer not to relive.

Jamie: So what do we do?

Doug: As I see it, three things. First we have to develop—or borrow—a change control process.

Jamie: You mean how people request changes?

Vinod: Yeah, but also how we evaluate the change, decide when to do it (if that's what we decide), and how we keep records of what's affected by the change.

Doug: Second, we've got to get a really good SCM tool for change and version control.

Jamie: We can build a database for all of our work products.

Vinod: They're called SCIs in this context, and most good tools provide some support for that.

Doug: That's a good start, now we have to . . .

Jamie: Uh, Doug, you said there were three things . . .

Doug (smiling): Third—we've all got to commit to follow the change management process and use the tools—no matter what, okay?

29.3.4 Impact Management

A web of software work product interdependencies must be considered every time a change is made. *Impact management* encompasses the work required to properly understand these interdependencies and control their effects on other SCIs (and the people who are responsible for them).

Impact management is accomplished with three actions [Sou08]. First, an *impact network* identifies the members of a software team (and other stakeholders) who might effect or be affected by changes that are made to the software. A clear definition of the software architecture (Chapter 13) assists greatly in the creation of an impact network. Next, *forward impact management* assesses the impact

of your own changes on the members of the impact network and then informs members of the impact of those changes. Finally, *backward impact management* examines changes that are made by other team members and their impact on your work and incorporates mechanisms to mitigate the impact.

29.3.5 Configuration Audit

Identification, version control, and change control help you to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can a software team ensure that the change has been properly implemented? The answer is twofold: (1) technical reviews and (2) the software configuration audit.

The technical review (Chapter 20) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A technical review should be conducted for all but the most trivial changes.

A *software configuration audit* complements the technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

? What are the primary questions that we ask during a configuration audit?

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a technical review. However, when SCM is a formal activity, the configuration audit is conducted separately by the quality assurance group. Such formal configuration audits also ensure that the correct SCIs (by version) have been incorporated into a specific build and that all documentation is up to date and consistent with the version that has been built.

29.3.6 Status Reporting

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

ADVICE

Develop a "need to know" list for every configuration object and keep it up to date. When a change is made, be sure that everyone on the list is notified.

The flow of information for configuration status reporting (CSR) is illustrated in Figure 29.5. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an online database or website, so that software developers or support staff can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

SOFTWARE TOOLS

SCM Support

Objective: SCM tools provide support to one or more of the process activities discussed in Section 29.3.

Mechanics: Most modern SCM tools work in conjunction with a repository (a database system) and provide mechanisms for identification, version and change control, auditing, and reporting.

Representative Tools:[8]

Software Change Manager, distributed by Computer Associates (**http://www.ca.com/us/products/detail/ca-change-manager-enterprise-workbench.aspx**), is a multiplatform SCM system.

ClearCase, developed by Rational, provides a family of SCM functions (**http://www-03.ibm.com/software/products/us/en/clearcase**).

Serena Dimensions CMF, distributed by Serena (**http://www.serena.com/index.php/en/products/dimensions-cm/**), provides a full set of SCM tools that are applicable for both conventional software and WebApps.

Allura, distributed by SourceForge Inc. (**http://sourceforge.net/p/allura/wiki/Allura%20Wiki/**), provides version management, build capabilities, issue/bug tracking, and many other management features.

SurroundSCM, developed by Seapine Software, provides complete change management capabilities (**www.seapine.com**).

Vesta, distributed by Compac, is a public-domain SCM system that can support both small (<10 KLOC) and large (10,000 KLOC) projects (**www.vestasys.org**).

A comprehensive list of commercial SCM tools and environments can be found at **www.cmtoday.com**.

29.4 CONFIGURATION MANAGEMENT FOR WEB AND MOBILEAPPS

Earlier in this book, we discussed the special nature of Web and MobileApps and the specialized methods[9] that are required to build them. Among the many characteristics that differentiate these applications from traditional software is the ubiquitous nature of change.

Web and mobile developers often use an iterative, incremental process model that applies many principles derived from agile software development (Chapter 5). Using this approach, an engineering team often develops an increment in

8 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

9 See [Pre08] for a comprehensive discussion of Web engineering methods.

a very short time period using a customer-driven approach. Subsequent increments add additional content and functionality, and each is likely to implement changes that lead to enhanced content, better usability, improved aesthetics, better navigation, enhanced performance, and stronger security. Therefore, in the agile world of Web and MobileApps, change is viewed somewhat differently.

If you're a member of a software team that builds Web or MobileApps, you must embrace change. And yet, a typical agile team eschews all things that appear to be process-heavy, bureaucratic, and formal. Software configuration management is often viewed (albeit incorrectly) to have these characteristics. This seeming contradiction is remedied not by rejecting SCM principles, practices, and tools, but rather by molding them to meet the special needs of Web and mobile projects.

29.4.1 Dominant Issues

? What impact does uncontrolled change have on a WebApp?

As Web and MobileApps become increasingly important to business survival and growth, the need for configuration management grows. Why? Because without effective controls, improper changes to these applications (recall that immediacy and continuous evolution are dominant attributes) can lead to: unauthorized posting of new product information, erroneous or poorly tested functionality that frustrates users, security holes that jeopardize internal company systems, and other economically unpleasant or even disastrous consequences.

The general strategies for software configuration management (SCM) described in this chapter are applicable, but tactics and tools must be adapted to conform to the unique nature of Web and MobileApps.

Four issues [Dar99] should be considered when developing tactics for WebApp configuration management.

Content. A typical WebApp contains a vast array of content—text, graphics, applets, scripts, audio/video files, forms, active page elements, tables, streaming data, and many others. The challenge is to organize this sea of content into a rational set of configuration objects (Section 29.1.4) and then establish appropriate configuration control mechanisms for these objects. One approach is to model the WebApp content using conventional data modeling techniques [Wik12], attaching a set of specialized properties to each object. The static/dynamic nature of each object and its projected longevity (e.g., temporary, fixed existence, or permanent object) are examples of properties that are required to establish an effective SCM approach. For example, if a content item is changed hourly, it has temporary longevity. The control mechanisms for this item would be different (less formal) from those applied for a forms component that is a permanent object.

People. Because a significant percentage of WebApp development continues to be conducted in an ad hoc manner, any person involved in the WebApp can (and often does) create content. Many content creators have no software engineering

background and are completely unaware of the need for configuration management. As a consequence, the application grows and changes in an uncontrolled fashion.

Scalability. The techniques and controls applied to a small WebApp do not scale upward well. It is not uncommon for a simple WebApp to grow significantly as interconnections with existing information systems, databases, data warehouses, and portal gateways are implemented. As size and complexity grow, small changes can have far-reaching and unintended effects that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale.

? How do I determine who has responsibility for WebApp CM?

Politics. Who "owns" a WebApp? This question is argued in companies large and small, and its answer has a significant impact on the management and control activities. The following questions [Dar99] help a software team understand the politics associated with Web engineering: Who assumes responsibility for the accuracy of the information on the website? Who ensures that quality control processes have been followed before information is published to the site? Who is responsible for making changes? Who assumes the cost of change? The answers to these questions help determine the people within an organization who must adopt a configuration management process for WebApps.

SCM techniques for MobileApps adopt many of the same principles that are applied for agile software development. In addition to the conventional SCM tasks discussed earlier in this chapter, change management for MobileApps must also consider the security implications of every change and its impact on a broad user base operating in a diverse platform environment.

The dramatic growth of app stores has changed the manner in which mobile software is deployed. Changes to a specific application can be promulgated widely in a matter of days, thereby requiring very careful analysis of impact across platforms before a new version of a MobileApp is placed in an app store for deployment.

In many instances, a conventional SCM process may be too cumbersome for WebApps and some MobileApps, but a new generation of *content management tools* that are specifically designed for these application areas have emerged over the past decade. These tools establish a process that acquires existing information (content objects), manages changes to the objects, structures it in a way that enables it to be presented to an end user, and then provides it to the client-side environment for display.

29.4.2 Configuration Objects

Web and MobileApps encompass a broad range of configuration objects—content objects (e.g., text, graphics, images, video, audio), functional components (e.g., scripts, applets), and interface objects (e.g., COM or CORBA for WebApps).

Objects can be identified (assigned file names) in any manner that is appropriate for the organization.

All content has format and structure. Internal file formats are dictated by the computing environment in which the content is stored. However, *rendering format* (often called *display format*) is defined by the aesthetic style and design rules established for the Web or MobileApp. *Content structure* defines a content architecture; that is, it defines the way in which content objects are assembled to present meaningful information to an end user. Boiko [Boi04] defines structure as "maps that you lay over a set of content chunks [objects] to organize them and make them accessible to the people who need them."

29.4.3 Content Management

Quote:

"Content management is an antidote to today's information frenzy."

Bob Boiko

Content management is related to configuration management in the sense that a content management system (CMS) establishes a process (supported by appropriate tools) that acquires existing content (from a broad array of WebApp and/or MobileApp configuration objects), structures it in a way that enables it to be presented to an end user, and then provides it to the client-side environment for display.

The most common use of a content management system occurs when a dynamic application is built. Dynamic Web or MobileApps create pages "on the fly." That is, the user typically queries the app requesting specific information. The app queries a server-side database, formats the information accordingly, and presents it to the user. For example, a music store (e.g., Apple iTunes) provides hundreds of thousands of tracks for sale. When a user requests a music track, a database is queried and a variety of information about the artist, the CD (e.g., its cover image or graphics), the musical content, and sample audio are all downloaded and configured into a standard content template. The resultant page is built on the server side and passed to the client side for examination by the end user. A generic representation for WebApps is shown in Figure 29.6.

In the most general sense, a CMS "configures" content for the end user by invoking three integrated subsystems: a collection subsystem, a management subsystem and a publishing subsystem [Boi04].

KEY POINT

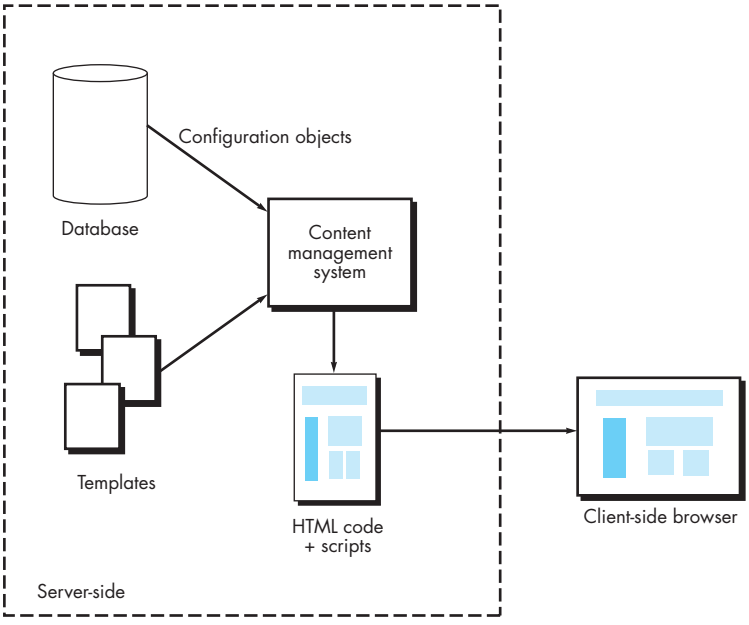
The collection subsystem encompasses all actions required to create, acquire, and/or convert content into a form that can be presented on the client side.

The collection subsystem. Content is derived from data and information that must be created or acquired by a content developer. The *collection subsystem* encompasses all actions required to create and/or acquire content, and the technical functions that are necessary to (1) convert content into a form that can be represented by a mark-up language (e.g., HTML, XML), and (2) organize content into packets that can be displayed effectively on the client side.

Content creation and acquisition (often called *authoring*) commonly occurs in parallel with other development activities and is often conducted by nontechnical content developers. This activity combines elements of creativity and research

FIGURE 29.6

Content management system



and is supported by tools that enable the content author to characterize content in a manner that can be standardized for use within the Web or MobileApp.

Once content exists, it must be converted to conform to the requirements of a CMS. This implies stripping raw content of any unnecessary information (e.g., redundant graphical representations), formatting the content to conform to the requirements of the CMS, and mapping the results into an information structure that will enable it to be managed and published.

KEY POINT

The management subsystem implements a repository for all content. Configuration management is performed within this subsystem.

The management subsystem. Once content exists, it must be stored in a repository, cataloged for subsequent acquisition and use, and labeled to define (1) current status (e.g., is the content object complete or in development?), (2) the appropriate version of the content object, and (3) related content objects. Therefore, the *management subsystem* implements a repository that encompasses the following elements:

- *Content database*—The information structure that has been established to store all content objects.
- *Database capabilities*—Functions that enable the CMS to search for specific content objects (or categories of objects), store and retrieve objects, and manage the file structure that has been established for the content.
- *Configuration management functions*—The functional elements and associated workflow that support content object identification, version control, change management, change auditing, and reporting.

In addition to these elements, the management subsystem implements an administration function that encompasses the metadata and rules that control the overall structure of the content and the manner in which it is supported.

KEY POINT

The publishing subsystem extracts content from the repository and delivers it to client-side browsers.

The publishing subsystem. Content must be extracted from the repository, converted to a form that is amenable to publication, and formatted so that it can be transmitted to client-side browsers. The publishing subsystem accomplishes these tasks using a series of templates. Each *template* is a function that builds a publication using one of three different components [Boi04]:

- *Static elements*—Text, graphics, media, and scripts that require no further processing are transmitted directly to the client side.
- *Publication services*—Function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
- *External services*—Access to external corporate information infrastructure such as enterprise data or "back-room" applications.

A content management system that encompasses each of these subsystems is applicable for major Web and mobile projects. However, the basic philosophy and functionality associated with a CMS are applicable to all dynamic applications.

SOFTWARE TOOLS

Content Management

Objective: To assist software engineers and content developers in managing content that is incorporated into WebApps.

Mechanics: Tools in this category enable Web engineers and content providers to update WebApp content in a controlled manner. Most establish a simple file management system that assigns page-by-page update and editing permissions for various types of WebApp content. Some maintain a versioning system so that a previous version of content can be archived for historical purposes.

Representative Tools:[10]

Open Text Web Experience Management, developed by Vignette (**http://www.opentext.com/global/products/web-content-management/web-experience-management/opentext-web-experience-management.htm**), is a suite of enterprise content management tools.

ektron-CMS300, developed by ektron (**www.ektron.com**), is a suite of tools that provide content management capabilities as well as Web development tools.

OmniUpdate, developed by WebsiteASP, Inc. (**www.omniupdate.com**), is a tool that allows authorized content providers to develop controlled updates to specified WebApp content.

Additional information on SCM and content management tools for Web engineering can be found at one or more of the following websites: *WebDeveloper* (**www.webdeveloper.com**), *Developer Shed* (**www.devshed.com**), *webknowhow.net* (**www.webknowhow.net**), or *WebReference* (**www.webreference.com**).

10 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

29.4.4 Change Management

The workflow associated with change control for conventional software (Section 29.3.3) is generally too ponderous for WebApp and mobile software development. It is unlikely that the change request, change report, and engineering change order sequence can be achieved in an agile manner that is acceptable for many Web and MobileApp development projects. How then do we manage a continuous stream of changes requested for content and functionality?

To implement effective change management within the "code and go" philosophy that continues to dominate much of Web and mobile development, the conventional change control process must be modified. Each change should be categorized into one of four classes:

Class 1—A content or function change that corrects an error or enhances local content or functionality.

Class 2—A content or function change that has an impact on other content objects or functional components.

Class 3—A content or function change that has a broad impact across an app (e.g., major extension of functionality, significant enhancement or reduction in content, major required changes in navigation).

Class 4—A major design change (e.g., a change in interface design or navigation approach) that will be immediately noticeable to one or more categories of user.

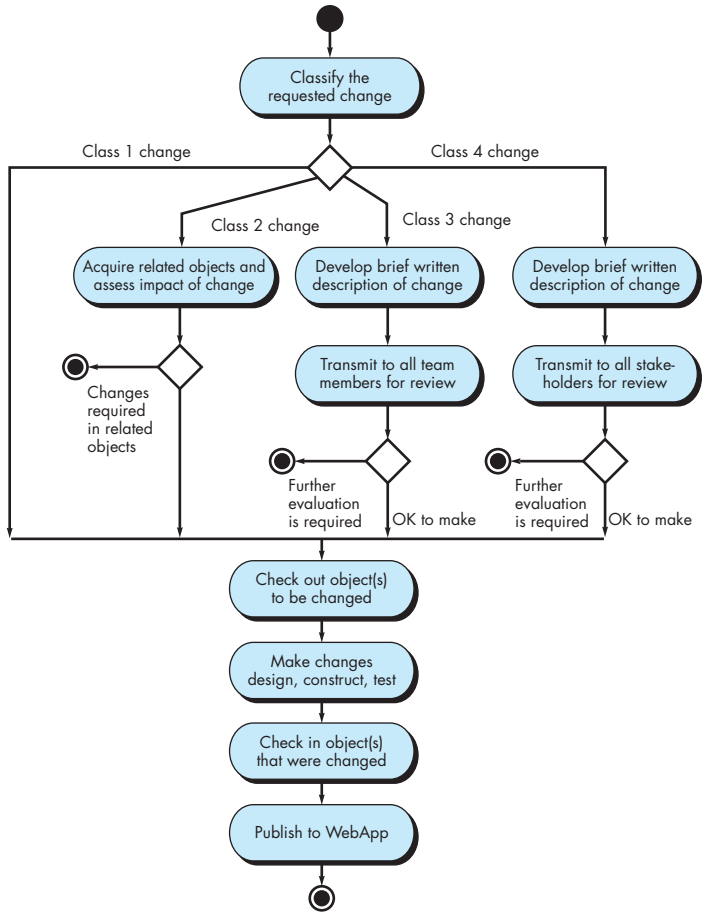
Once the requested change has been categorized, it can be processed according to the algorithm shown in Figure 29.7 for WebApps but equally applicable for MobileApps.

Referring to the figure, class 1 and 2 changes are treated informally and are handled in an agile manner. For a class 1 change, you would evaluate the impact of the change, but no external review or documentation is required. As the change is made, standard check-in and check-out procedures are enforced by configuration repository tools. For class 2 changes, you should review the impact of the change on related objects (or ask other developers responsible for those objects to do so). If the change can be made without requiring significant changes to other objects, modification occurs without additional review or documentation. If substantive changes are required, further evaluation and planning are necessary.

Class 3 and 4 changes are also treated in an agile manner, but some descriptive documentation and more formal review procedures are required. A *change description*—describing the change and providing a brief assessment of the impact of the change—is developed for class 3 changes. The description is distributed to all members of the team who review it to better assess its impact. A change description is also developed for class 4 changes, but in this case, the review is conducted by all stakeholders.

FIGURE 29.7

Managing changes for WebApps



SOFTWARE TOOLS

Change Management

Objective: To assist Web engineers and content developers in managing changes as they are made to WebApp configuration objects.

Mechanics: Tools in this category were originally developed for conventional software but can be adapted for use by Web engineers and content developers to make controlled changes to WebApps. They support automated check-in and check-out, version control and rollback, reporting, and other SCM functions.

Representative Tools:[11]

Dimension CM, developed by Serena (**http://www.serena.com/index.php/en/products/dimensions-cm/**), is one of a suite of change management tools that provide complete SCM capabilities.

ClearCase, developed by Rational (**http://www-03.ibm.com/software/products/us/en/clearcase**), is a suite of tools that provide full configuration management capabilities for WebApps.

PTC Integrity, developed by PTC (**http://www.mks.com/platform/our-product**), is an SCM tool that can be integrated with selected development environments.

11 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

29.4.5 Version Control

As a Web or MobileApp evolves through a series of increments, a number of different versions may exist at the same time. One version (the current operational app) is available via the Internet for end users; another version (the next app increment) may be in the final stages of testing prior to deployment; a third version is in development and represents a major update in content, interface aesthetics, and functionality. Configuration objects must be clearly defined so that each can be associated with the appropriate version. In addition, control mechanisms must be established. Dreilinger [Dre99] discusses the importance of version (and change) control when he writes:

> In an *uncontrolled* site where multiple authors have access to edit and contribute, the potential for conflict and problems arises—more so when these authors work from different offices at different times of day and night. You may spend the day improving the file *index.html* for a customer. After you've made your changes, another developer who works at home after hours, or in another office, may spend the night uploading their own newly revised version of the file *index.html*, completely overwriting your work with no way to get it back!

It's likely that you've experienced a similar situation. To avoid it, a version control process is required.

1. *A central repository for the Web or MobileApp project should be established.* The repository will hold current versions of all configuration objects (content, functional components, and others).
2. *Each Web engineer creates his or her own working folder.* The folder contains those objects that are being created or changed at any given time.
3. *The clocks on all developer workstations should be synchronized.* This is done to avoid overwriting conflicts when two developers make updates that are very close to one another in time.
4. *As new configuration objects are developed or existing objects are changed, they are imported into the central repository.* The version control tool (see discussion of CVS in the sidebar) will manage all check-in and check-out functions from the working folders of each developer. The tool will also provide automatic e-mail updates to all interested parties when changes to the repository are made.
5. *As objects are imported or exported from the repository, an automatic, time-stamped log message is made.* This provides useful information for auditing and can become part of an effective reporting scheme.

The version control tool maintains different versions of the app and can revert to an older version if required.

29.4.6 Auditing and Reporting

In the interest of agility, the auditing and reporting functions are deemphasized during the development of Web or MobileApps.[12] However, they are not eliminated altogether. All objects that are checked into or out of the repository are recorded in a log that can be reviewed at any point in time. A complete log report can be created so that all members of the team have a chronology of changes over a defined period of time. In addition, an automated e-mail notification (addressed to those developers and stakeholders who have interest) can be sent every time an object is checked in or out of the repository.

INFO

SCM Standards

The following list of SCM standards (extracted in part from **www.12207.com**) is reasonably comprehensive:

IEEE Standards	**standards.ieee.org/catalog/olis/**
IEEE 828	Software Configuration Management Plans
IEEE 1042	Software Configuration Management
ISO Standards	**http://www.iso.org/iso/home**
ISO 10007-1995	Quality Management, Guidance for CM
ISO/IEC 12207	Information Technology-Software Life Cycle Processes
ISO/IEC TR 15271	Guide for ISO/IEC 12207
ISO/IEC TR 15846	Software Engineering-Software Life Cycle Process-Configuration Management for Software Order
EIA Standards	**www.eia.org/**
EIA 649	National Consensus Standard for Configuration Management
EIA CMB4-1A	Configuration Management Definitions for Digital Computer Programs
EIA CMB4-2	Configuration Identification for Digital Computer Programs
EIA CMB4-3	Computer Software Libraries
EIA CMB4-4	Configuration Change Control for Digital Computer Programs
EIA CMB6-1C	Configuration and Data Management References Order
EIA CMB6-3	Configuration Identification
EIA CMB6-4	Configuration Control
EIA CMB6-5	Textbook for Configuration Status Accounting
EIA CMB7-1	Electronic Interchange of Configuration Management Data
U.S. Military Standards	
DoD MIL STD-973	Configuration Management
MIL-HDBK-61	Configuration Management Guidance
Other Standards	
DO-178B	Guidelines for the Development of Aviation Software
ESA PSS-05-09	Guide to Software Configuration Management
AECL CE-1001-STD rev. 1	Standard for Software Engineering of Safety Critical Software

12 This is beginning to change. There is an increasing emphasis on SCM as one element of application security [Sar06]. By providing a mechanism for tracking and reporting every change made to every application object, a change management tool can provide valuable protection against malicious changes.

DOE SCM checklist:	**http://energy.gov/cio/downloads/software-quality-systems-engineering-program-software-configuration-management**
BS-6488	British Std., Configuration Management of Computer-Based Systems
Best Practice—UK	Office of Government Commerce: **http://www.cabinetoffice.gov.uk/content/office-government-commerce-ogc**
CMII	Institute of CM Best Practices: **www.icmhq.com**

A *Configuration Management Resource Guide* provides complementary information for those interested in CM processes and practice. It is available at **http://cmpic.com/cmresourceguide.htm**.

29.5 Summary

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All work products created as part of software engineering become part of a software configuration. The configuration is organized in a manner that enables orderly control of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items, that are produced as a result of some software engineering activity. In addition to software engineering work products, the development environment that is used to create software can also be placed under configuration control. All SCIs are stored within a repository that implements a set of mechanisms and data structures to ensure data integrity, provide integration support for other software tools, support information sharing among all members of the software team, and implement functions in support of version and change control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Version control is the set of procedures and tools for managing the use of these objects.

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

Configuration management for Web and MobileApps is similar in most respects to SCM for conventional software. However, each of the core SCM tasks

should be streamlined to make it as lean as possible, and special provisions for content management must be implemented.

Problems and Points to Ponder

29.1. Why is the First Law of System Engineering true? Provide specific examples for each of the four fundamental reasons for change.

29.2. What are the four elements that exist when an effective SCM system is implemented? Discuss each briefly.

29.3. Discuss the reasons for baselines in your own words.

29.4. Assume that you're the manager of a small project. What baselines would you define for the project and how would you control them?

29.5. Design a project database (repository) system that would enable a software engineer to store, cross-reference, trace, update, change, and so forth all important software configuration items. How would the database handle different versions of the same program? Would source code be handled differently than documentation? How will two developers be precluded from making different changes to the same SCI at the same time?

29.6. Research an existing SCM tool and describe how it implements control for versions, variants, and configuration objects in general.

29.7. The relations <part-of> and <interrelated> represent simple relationships between configuration objects. Describe five additional relationships that might be useful in the context of an SCM repository.

29.8. Research an existing SCM tool and describe how it implements the mechanics of version control. Alternatively, read two or three papers on SCM and describe the different data structures and referencing mechanisms that are used for version control.

29.9. Develop a checklist for use during configuration audits.

29.10. What is the difference between an SCM audit and a technical review? Can their function be folded into one review? What are the pros and cons?

29.11. Briefly describe the differences between SCM for conventional software and SCM for Web or MobileApps.

29.12. What is content management? Use the Web to research the features of a content management tool and provide a brief summary.

Further Readings and Information Sources

Among the more recent SCM offerings are Aiello and his colleagues (*Configuration Management Best Practices: Practical Methods That Work in the Real World,* Addison-Wesley, 2010), Moreira (*Adapting Configuration Management for Agile Teams: Balancing Sustainability and Speed,* Wiley, 2009), Duvall and his colleagues (*Continuous Integration: Improving Software Quality and Reducing Risk,* Addison-Wesley, 2007), Leon (*Software Configuration Management Handbook,* 2nd ed., Artech House Publishers, 2005), Maraia (*The Build Master: Microsoft's Software Configuration Management Best Practices,* Addison-Wesley, 2005), Keyes (*Software Configuration Management,* Auerbach, 2004), and Hass (*Configuration Management Principles and Practice,* Addison-Wesley, 2002). Each of these books presents the entire SCM process in substantial detail. Moreira (*Software Configuration Management Implementation Roadmap,* Wiley, 2004) presents a unique how-to guide for those who must

implement SCM within an organization. Lyon (*Practical CM III: Best Practices for the 21st Century,* Raven Publishing, 2013, available at **www.configuration.org**) has written a comprehensive guide for the CM professional that includes pragmatic guidelines for implementing every aspect of a configuration management system (updated yearly). Girod and Shpichko (*IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations*, Packt, 2011) and Bellagio and Mulligan present SCM within the context of one of the more popular SCM tools.

Berczuk and Appleton (*Software Configuration Management Patterns,* Addison-Wesley, 2003) present a variety of useful patterns that assist in understanding SCM and implementing effective SCM systems. Brown et al. (*Anti-Patterns and Patterns in Software Configuration Management,* Wiley, 1999) discuss the things not to do (anti-patterns) when implementing an SCM process and then consider their remedies. Humble and Fowler (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation,* Addison-Wesley, 2010) and Bays (*Software Release Methodology,* Prentice Hall, 1999) focuses on the mechanics of "successful product release," an important complement to effective SCM.

As WebApps have become more dynamic, content management has become an essential topic for Web engineers. Books by Rockley and Cooper (*Managing Enterprise Content: A Unified Content Strategy,* 2nd ed., New Riders, 2012), Jenkins and his colleagues (*Managing Content in the Cloud—Enterprise Content Management 2.0,* Open Text Corporation, 2010) and (*Enterprise Content Management Methods,* Open Text Corporation, 2005), White (*The Content Management Handbook,* Curtin University Books, 2005), Boiko [Boi04], Mauthe and Thomas (*Professional Content Management Systems,* Wiley, 2004), Addey and his colleagues (*Content Management Systems,* Glasshaus, 2003), Hackos (*Content Management for Dynamic Web Delivery,* Wiley, 2002), and Nakano (*Web Content Management,* Addison-Wesley, 2001) present worthwhile treatments of the subject.

In addition to generic discussions of the topic, Halvorson and Bach (*Content Strategy for the Web,* 2nd ed., New Riders, 2012), Hauschildt (*CMS Made Simple 1.6: Beginners' Guide*, Packt, 2010), Lim and his colleagues (*Enhancing Microsoft Content Management Server with ASP.NET 2.0,* Packt Publishing, 2006), Ferguson (*Creating Content Management Systems in Java,* Charles River Media, 2006), IBM Redbooks (*IBM Workplace Web Content Management for Portal 5.1 and IBM Workplace Web Content Management 2.5,* Vivante, 2006), Fritz and his colleagues (*Typo3: Enterprise Content Management,* Packt Publishing, 2005), and Forta (*Reality ColdFusion: Intranets and Content Management,* Pearson Education, 2002) present content management within the context of specific tools and languages.

A wide variety of information sources on software configuration management and content management is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

30 Product Metrics

Key Concepts

architectural design metrics 663
class-oriented metrics 667
function point (FP) . 659
goal/question/metric (GQM) 656
indicator 655
measure 654
measurement 654
measurement principles 656

A key element of any engineering process is measurement. You can use measures to better understand the attributes of the models that you create and to assess the quality of the engineered products or systems that you build. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Because software measures and metrics are often indirect, they are open to debate. Fenton [Fen91] addresses this issue when he states:

> Measurement is the process by which numbers or symbols are assigned to the attributes of entities in the real world in such a way as to define them according to clearly defined rules . . . In the physical sciences, medicine, economics, and more recently the social sciences, we are now able to measure attributes that we previously thought to be unmeasurable . . . Of course, such measurements are not

Quick Look

What is it? By its nature, engineering is a quantitative discipline. Product metrics help software engineers gain insight into the design and construction of the software they build by focusing on specific, measurable attributes of software engineering work products.

Who does it? Software engineers use product metrics to help them build higher-quality software.

Why is it important? There will always be a qualitative element to the creation of computer software. The problem is that qualitative assessment may not be enough. You need objective criteria to help guide the design of data, architecture, interfaces, and components. When testing, you need quantitative guidance that will help in the selection of test cases and their targets. Product metrics provide a basis from which analysis, design, coding, and testing can be conducted more objectively and assessed more quantitatively.

What are the steps? The first step in the measurement process is to derive the software measures and metrics that are appropriate for the representation of software that is being considered. Next, data required to derive the formulated metrics are collected. Once computed, appropriate metrics are analyzed based on preestablished guidelines and past data. The results of the analysis are interpreted to gain insight into the quality of the software, and the results of the interpretation lead to modification of requirements and design models, source code, or test cases. In some instances, it may also lead to modification of the software process itself.

What is the work product? Product metrics that are computed from data collected from the requirements and design models, source code, and test cases.

How do I ensure that I've done it right? You should establish the objectives of measurement before data collection begins, defining each product metric in an unambiguous manner. Define only a few metrics and then use them to gain insight into the quality of a software engineering work product.

metrics for object-oriented design . . . 666
metrics
attributes of . . 657
component-level 671
design 663
MobileApps . . . 672
requirements model 659
source code . . . 675
testing. 676
user interface design 672
WebApps 672

> as refined as many measurements in the physical sciences . . . , but they exist [and important decisions are made based on them]. We feel that the obligation to attempt to "measure the unmeasurable" in order to improve our understanding of particular entities is as powerful in software engineering as in any discipline.

But some members of the software community continue to argue that software is "unmeasurable" or that attempts at measurement should be postponed until we better understand software and the attributes that should be used to describe it. This is a mistake.

Although product metrics for computer software are imperfect, they can provide you with a systematic way to assess quality based on a set of clearly defined rules. They also provide you with on-the-spot, rather than after-the-fact, insight. This enables you to discover and correct potential problems before they become catastrophic defects.

In this chapter, we present measures that can be used to assess the quality of the product as it is being engineered. These measures of internal product attributes provide you with a real-time indication of the efficacy of the requirements, design, and code models; the effectiveness of test cases; and the overall quality of the software to be built.

30.1 A Framework for Product Metrics

Quote:

"A science is as mature as its measurement tools."

Louis Pasteur

Measurement assigns numbers or symbols to attributes of entities in the real word. To accomplish this, a measurement model encompassing a consistent set of rules is required. Although the theory of measurement (e.g., [Kyb84]) and its application to computer software (e.g., [Zus97]) are topics that are beyond the scope of this book, it is worthwhile to establish a fundamental framework and a set of basic principles that guide the definition of product metrics for software.

30.1.1 Measures, Metrics, and Indicators

? What's the difference between a measure and a metric?

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. Because *measure* can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a *measure* provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. *Measurement* is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terminology* [IEE93b] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g.,

a number of component reviews and unit tests are investigated to collect measures of the number of errors for each). A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test).

KEY POINT

An indicator is a metric or metrics that provide insight into the process, the product, or the project.

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the product to make things better.

30.1.2 The Challenge of Product Metrics

Over the past four decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton [Fen94] characterizes this research as a search for "the impossible holy grail." Although dozens of complexity measures have been proposed [Zus90], each takes a somewhat different view of what complexity is and what attributes of a system lead to complexity. By analogy, consider a metric for evaluating an attractive car. Some observers might emphasize body design; others might consider mechanical characteristics; still others might tout cost, or performance, or the use of alternative fuels, or the ability to recycle when the car is junked. Since any one of these characteristics may be at odds with others, it is difficult to derive a single value for "attractiveness." The same problem occurs with computer software.

WebRef

Voluminous information on product metrics has been compiled by the U.S. Department of Homeland Security at **https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/measurement.html**.

Yet there is a need to measure and control software complexity. And if a single value of this quality metric is difficult to derive, it should be possible to develop measures of different internal program attributes (e.g., effective modularity, functional independence, and other attributes discussed in Chapter 12). These measures and the metrics derived from them can be used as independent indicators of the quality of requirements and design models. But here again, problems arise. Fenton [Fen94] notes this when he states: "The danger of attempting to find measures which characterize so many different attributes is that inevitably the measures have to satisfy conflicting aims. This is counter to the representational theory of measurement." Although Fenton's statement is correct, many people argue that product measurement conducted during the early stages of the software process provides software engineers with a consistent and objective mechanism for assessing quality.[1]

1 Although criticism of specific metrics is common in the literature, many critiques focus on esoteric issues and miss the primary objective of metrics in the real world: to help the software engineer establish a systematic and objective way to gain insight into his or her work and to improve product quality as a result.

30.1.3 Measurement Principles

Quote:

"Just as temperature measurement began with an index finger . . . and grew to sophisticated scales, tools and techniques, so too is software measurement maturing."

Shari Pfleeger

Before we introduce a series of product metrics that (1) assist in the evaluation of the analysis and design models, (2) provide an indication of the complexity of procedural designs and source code, and (3) facilitate the design of more effective testing, it is important for you to understand basic measurement principles. Roche [Roc94] suggests a measurement process that can be characterized by five activities: formulation, collection, analysis, interpretation, and feedback. Software metrics will be useful only if they are characterized effectively and validated so that their worth is proven. The following principles [Let03b] are representative of many that can be proposed for metrics characterization and validation:

- *A metric should have desirable mathematical properties.* That is, the metric's value should be in a meaningful range (e.g., 0 to 1, where 0 truly means absence, 1 indicates the maximum value, and 0.5 represents the "halfway point"). Also, a metric that purports to be on a rational scale should not be composed of components that are only measured on an ordinal scale.
- *When a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner.*
- *Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions.* A metric should measure the factor of interest, independently of other factors. It should "scale up" to large systems and work in a variety of programming languages and system domains.

Although formulation, characterization, and validation are critical, collection and analysis are the activities that drive the measurement process. Roche [Roc94] suggests the following guidelines for these activities: (1) whenever possible, data collection and analysis should be automated; (2) valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics (e.g., whether the level of architectural complexity correlates with the number of defects reported in production use); and (3) interpretative guidelines and recommendations should be established for each metric.

WebRef

A useful discussion of GQM can be found at **https://www.thecsiac.com/resources/ref_documents/software-acquisition-gold-practice-goal-question-metric-gqm-approach**.

30.1.4 Goal-Oriented Software Measurement

The *Goal/Question/Metric* (GQM) paradigm has been developed by Basili and Weiss [Bas84] as a technique for identifying meaningful metrics for any part of the software process. GQM emphasizes the need to (1) establish an explicit measurement *goal* that is specific to the process activity or product characteristic that is to be assessed, (2) define a set of *questions* that must be answered in order to achieve the goal, and (3) identify well-formulated *metrics* that help to answer these questions.

A *goal definition template* [Bas94] can be used to define each measurement goal. The template takes the form:

> **Analyze** {the name of activity or attribute to be measured} **for the purpose of** {the overall objective of the analysis}[2] **with respect to** {the aspect of the activity or attribute that is considered} **from the viewpoint of** {the people who have an interest in the measurement} **in the context of** {the environment in which the measurement takes place}.

As an example, consider a goal definition template for *SafeHome*:

> **Analyze** the *SafeHome* software architecture **for the purpose of** evaluating architectural components **with respect to** the ability to make *SafeHome* more extensible **from the viewpoint of** the software engineers performing the work **in the context of** product enhancement over the next three years.

With a measurement goal explicitly defined, a set of questions is developed. Answers to these questions help the software team (or other stakeholders) to determine whether the measurement goal has been achieved. Among the questions that might be asked are:

Q_1: Are architectural components characterized in a manner that compartmentalizes function and related data?

Q_2: Is the complexity of each component within bounds that will facilitate modification and extension?

Each of these questions should be answered quantitatively, using one or more measures and metrics. For example, a metric that provides an indication of the cohesion (Chapter 12) of an architectural component might be useful in answering Q_1. Metrics discussed later in this chapter might provide insight for Q_2. In every case, the metrics that are chosen (or derived) should conform to the measurement principles discussed in Section 30.1.3 and the measurement attributes discussed in Section 30.1.5.

ADVICE

Experience indicates that a product metric will be used only if it is intuitive and easy to compute. If dozens of "counts" have to be made, and complex computations are required, it is unlikely that the metric will be widely adopted.

30.1.5 The Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer. Some demand measurement that is too complex, others are so esoteric that few real-world professionals have any hope of understanding them, and others violate the basic intuitive notions of what high-quality software really is.

Ejiogu [Eji91] defines a set of attributes that should be encompassed by effective software metrics. It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time.The metric should satisfy the engineer's intuitive notions about the product attribute under

2 van Solingen and Berghout [Sol99] suggest that the objective is almost always "understanding, controlling or improving" the process activity or product attribute.

consideration (e.g., a metric that measures module cohesion should increase in value as the level of cohesion increases). The metric should always yield results that are unambiguous. The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units. For example, multiplying people on the project teams by programming language variables in the program results in a suspicious mix of units that are not intuitively persuasive. Metrics should be based on the requirements model, the design model, or the structure of the program itself. They should not be dependent on the vagaries of programming language syntax or semantics. Finally, the metric should provide you with information that can lead to a higher-quality end product.

Although many software metrics satisfy all of these attributes, some commonly used metrics may fail to satisfy one or two of them. One example is the function point (discussed in Section 30.2.1)—a measure of the "functionality" delivered by the software. It can be argued that an independent third party may not be able to derive the same function point value as a colleague using the same information about the software.[3] Should we therefore reject the function point measure? The answer is of course not! FP provides useful insight and therefore provides distinct value, even if it fails to satisfy one attribute perfectly.

SafeHome

Debating Product Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team who are continuing work of component-level design and test-case design.

The conversation:

Vinod: Doug [Doug Miller, software engineering manager] told me that we should all use product metrics, but he was kind of vague. He also said that he wouldn't push the matter . . . that using them was up to us.

Jamie: That's good, 'cause there's no way I have time to start measuring stuff. We're fighting to maintain the schedule as it is.

Ed: I agree with Jamie. We're up against it, here . . . no time.

Vinod: Yeah, I know, but there's probably some merit to using them.

Jamie: I'm not arguing that, Vinod, it's a time thing . . . and I for one don't have any to spare.

Vinod: But what if measuring saves you time?

Ed: Wrong, it takes time and like Jamie said . . .

Vinod: No, wait . . . what it saves us is time?

Jamie: How?

Vinod: Rework . . . that's how. If a measure we use helps us to avoid one major or even moderate problem, and that saves us from having to rework a part of the system, we save time. No?

Ed: It's possible, I suppose, but can you guarantee that some product metric will help us find a problem?

Vinod: Can you guarantee that it won't?

Jamie: So what are you proposing?"

Vinod: I think we should select a few design metrics, probably class-oriented, and use them as part of our review process for every component we develop.

Ed: I'm not real familiar with class-oriented metrics.

Vinod: I'll spend some time checking them out and make a recommendation . . . okay with you guys?

[Ed and Jamie nod without much enthusiasm.]

3 An equally vigorous counterargument can be made. Such is the nature of software metrics.

30.2 Metrics for the Requirements Model

Technical work in software engineering begins with the creation of the requirements model. It is at this stage that requirements are derived and a foundation for design is established. Therefore, product metrics that provide insight into the quality of the analysis model are desirable.

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics that are often used for project estimation and apply them in this context. These metrics examine the requirements model with the intent of predicting the "size" of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort.

30.2.1 Function-Based Metrics

WebRef

Much useful information about function points can be obtained at **www.ifpug.org and http://www.functionpoint.com/**.

The *function point* (FP) *metric* can be used effectively as a means for measuring the functionality delivered by a system.[4] Using historical data, the FP metric can then be used to (1) estimate the cost or effort required to design, code, and test the software; (2) predict the number of errors that will be encountered during testing; and (3) forecast the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner:[5]

> **Number of external inputs (EIs).** Each *external input* originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update *internal logical files* (ILFs). Inputs should be distinguished from inquiries, which are counted separately.
>
> **Number of external outputs (EOs).** Each *external output* is derived data within the application that provides information to the user. In this context external output refers to reports, screens, error messages, and the like. Individual data items within a report are not counted separately.
>
> **Number of external inquiries (EQs).** An *external inquiry* is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).

4 Hundreds of books, papers, and articles have been written on FP metrics. A worthwhile bibliography can be found at [IFP05].

5 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP01] for more details.

FIGURE 30.1

Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		3	3	4	6	=	
External Outputs (EOs)		3	4	5	7	=	
External Inquiries (EQs)		3	3	4	6	=	
Internal Logical Files (ILFs)		3	7	10	15	=	
External Interface Files (EIFs)		3	5	7	10	=	
Count total							

Number of internal logical files (ILFs). Each *internal logical file* is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Number of external interface files (EIFs). Each *external interface file* is a logical grouping of data that resides external to the application but provides data that may be of use to the application.

Once these data have been collected, the table in Figure 30.1 is completed and a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] \tag{30.1}$$

where count total is the sum of all FP entries obtained from Figure 30.1.

The F_i (i = 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [Lon02]:

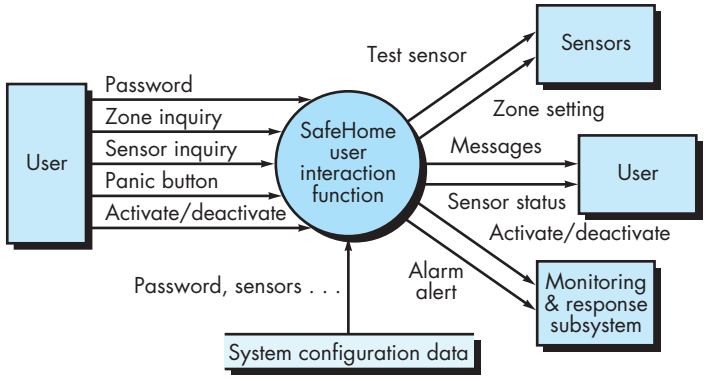
KEY POINT

Value adjustment factors are used to provide an indication of problem complexity.

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?

FIGURE 30.2

A flow model for *SafeHome* user interaction function



9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using an ordinal scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (30.1) and the weighting factors that are applied to information domain counts are determined empirically.

WebRef

An online FP calculator can be found at **http://groups.engin.umd.umich.edu/CIS/course.des/cis375/projects/fp99/main.html**.

To illustrate the use of the FP metric in this context, consider a simple flow diagram for a user interaction function within *SafeHome* software, represented in Figure 30.2. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.

The flow diagram is evaluated to determine a set of key information domain measures required for computation of the function point metric. Three external inputs—**password, panic button,** and **activate/deactivate**—are shown in the figure along with two external inquiries—**zone inquiry** and **sensor inquiry**. One ILF (**system configuration file**) is shown. Two external outputs (**messages** and **sensor status**) and four EIFs (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure 30.3.

FIGURE 30.3
Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	3	3	3	4	6	=	9
External Outputs (EOs)	2	3	4	5	7	=	8
External Inquiries (EQs)	2	3	3	4	6	=	6
Internal Logical Files (ILFs)	1	3	7	10	15	=	7
External Interface Files (EIFs)	4	3	5	7	10	=	20
Count total						→	50

The count total shown in Figure 30.3 must be adjusted using Equation (30.1). For the purposes of this example, we assume that $\Sigma (F_i)$ is 46 (a moderately complex product). Therefore,

$$FP = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Quote:

"Rather than just musing on what 'new metric' might apply . . . we should also be asking ourselves the more basic question, 'What will we do with metrics?'"

Michael Mah and Larry Putnam

Based on the projected FP value derived from the requirements model, the project team can estimate the overall implemented size of the *SafeHome* user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an object-oriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the requirements model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during requirements and design reviews and four errors per function point during unit and integration testing. These data can ultimately help you assess the completeness of your review and testing activities. Uemura and his colleagues [Uem99] suggest that function points can also be computed from UML class and sequence diagrams.

30.2.2 Metrics for Specification Quality

KEY POINT

By measuring characteristics of the specification, it is possible to gain quantitative insight into specificity and completeness.

Davis and his colleagues [Dav93] propose a list of characteristics that can be used to assess the quality of the requirements model and the corresponding requirements specification: *specificity* (lack of ambiguity), *completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision,* and *reusability.* In addition, the authors note that high-quality specifications are electronically stored; executable or at least interpretable; annotated by relative importance; and stable, versioned, organized, cross-referenced, and specified at the right level of detail.

Although many of these characteristics appear to be qualitative in nature, each can be represented using one or more metrics. [Dav93] For example, we assume that there are n_r requirements in a specification, such that

$$n_r = n_f + n_{nf}$$

where n_f is the number of functional requirements and n_{nf} is the number of nonfunctional (e.g., performance) requirements.

To determine the *specificity* (lack of ambiguity) of requirements, Davis and colleagues suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = \frac{n_{ui}}{n_r}$$

where n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

The *completeness* of functional requirements can be determined by computing the ratio

$$Q_2 = \frac{n_u}{[n_i \times n_s]}$$

where n_u is the number of unique function requirements, n_i is the number of inputs (stimuli) defined or implied by the specification, and n_s is the number of states specified. The Q_2 ratio measures the percentage of necessary functions that have been specified for a system. However, it does not address nonfunctional requirements. To incorporate these into an overall metric for completeness, you must consider the degree to which requirements have been validated:

$$Q_3 = \frac{n_c}{[n_c + n_{nv}]}$$

where n_c is the number of requirements that have been validated as correct and n_{nv} is the number of requirements that have not yet been validated.

Quote:

"Measure what is measurable, and what is not measurable, make measurable."

Galileo

30.3 Metrics for the Design Model

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them as indicators to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

KEY POINT

Metrics can provide insight into structural data and system complexity associated with architectural design.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.

30.3.1 Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture (Chapter 13) with an emphasis on the architectural structure and the

effectiveness of modules or components within the architecture. These metrics are "black box" in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass [Car90] define three software design complexity measures: structural complexity, data complexity, and system complexity.

For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module i is defined in the following manner:

$$S(i) = f^2_{out}(i) \tag{30.2}$$

where $f_{out}(i)$ is the fan-out[6] of module i.

Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = \frac{v(i)}{[f_{out}(i) + 1]} \tag{30.3}$$

where $v(i)$ is the number of input and output variables that are passed to and from module i.

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i) \tag{30.4}$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

Fenton [Fen91] suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to the call-and-return architecture in Figure 30.4, the following metrics can be defined:

$$\text{Size} = n + a$$

where n is the number of nodes and a is the number of arcs. For the architecture shown in Figure 30.4,

$$\text{Size} = 17 + 18 = 35$$

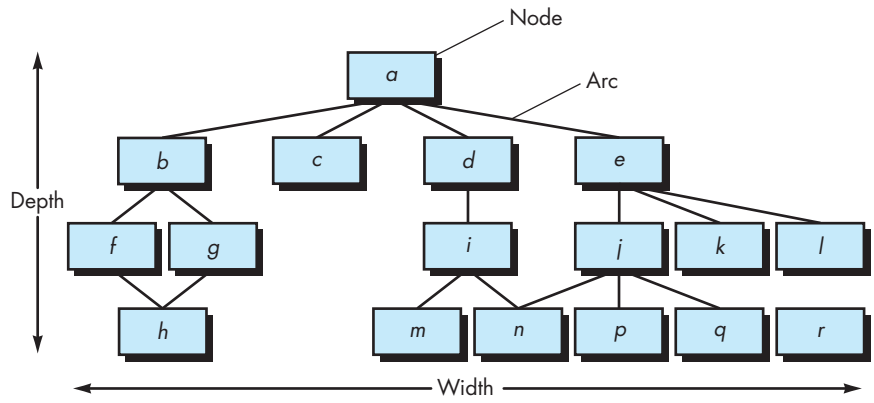
Depth = longest path from the root (top) node to a leaf node. For the architecture shown in Figure 30.4, depth = 4.

Width = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 30.4, width = 6.

6 *Fan-out* is defined as the number of modules immediately subordinate to module i; that is, the number of modules that are directly invoked by module i.

FIGURE 30.4

Morphology metrics



The arc-to-node ratio, $r = a/n$, measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in Figure 30.4, $r = 18/17 = 1.06$.

The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-2005 [IEE05], the Air Force uses information obtained from data and architectural design to derive a *design structure quality index* (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI [Cha89]:

S_1 = total number of modules defined in the program architecture

S_2 = number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S_2)

S_3 = number of modules whose correct function depends on prior processing

S_4 = number of database items (includes data objects and all attributes that define objects)

S_5 = total number of unique database items

S_6 = number of database segments (different records or individual objects)

S_7 = number of modules with a single entry and exit (exception processing is not considered to be a multiple exit)

Quote:

"Measurement can be seen as a detour. This detour is necessary because humans mostly are not able to make clear and objective decisions [without quantitative support]."

Horst Zuse

Once values S_1 through S_7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D_1, where D_1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow–oriented design or object-oriented design), then $D_1 = 1$, otherwise $D_1 = 0$.

Module independence: $D_2 = 1 - \left(\frac{S_2}{S_1}\right)$
Modules not dependent on prior processing: $D_3 = 1 - \left(\frac{S_3}{S_1}\right)$
Database size: $D_4 = 1 - \left(\frac{S_5}{S_4}\right)$
Database compartmentalization: $D_5 = 1 - \left(\frac{S_6}{S_4}\right)$
Module entrance/exit characteristic: $D_6 = 1 - \left(\frac{S_7}{S_1}\right)$

With these intermediate values determined, the DSQI is computed in the following manner:

$$\text{DSQI} = \Sigma\, w_i D_i \tag{30.5}$$

where $i = 1$ to 6, w_i is the relative weighting of the importance of each of the intermediate values, and $\Sigma\, w_i = 1$ (if all D_i are weighted equally, then $w_i = 0.167$).

The value of DSQI for past designs can be determined and compared to a design that is currently under development. If the DSQI is significantly lower than average, further design work and review are indicated. Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

30.3.2 Metrics for Object-Oriented Design

There is much about object-oriented design that is subjective—an experienced designer "knows" how to characterize an OO system so that it will effectively implement customer requirements. But, as an OO design model grows in size and complexity, a more objective view of the characteristics of the design can benefit both the experienced designer (who gains additional insight) and the novice (who obtains an indication of quality that would otherwise be unavailable).

In a detailed treatment of software metrics for OO systems, Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design. *Size* is defined by taking a static count of OO entities such as classes or operations, coupled with the depth of an inheritance tree. *Complexity* is defined in terms of structural characteristics by examining how classes of an OO design are interrelated to one another. *Coupling* is measured by counting physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects). *Sufficiency* is "the degree to which an abstraction [class] possesses the features required of it . . ." [Whi97]. *Completeness* determines whether a class delivers the set of properties that fully reflect the needs of the problem domain. *Cohesion* is determined be examining whether all operations work together to achieve a single, well-defined purpose. *Primitiveness* is the degree to which an operation is atomic—that is, the operation cannot be constructed out of a sequence of other operations contained within a class. Similarity determines the degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose. Volatility measures the likelihood that a change will occur.

In reality, product metrics for OO systems can be applied not only to the design model, but also to the requirements model. In the sections that follow, we discuss metrics that provide an indication of quality at the OO class level and the operation level. In addition, metrics applicable for project management and testing are also explored.

30.3.3 Class-Oriented Metrics—The CK Metrics Suite

The class is the fundamental unit of an OO system. Therefore, measures and metrics for an individual class, the class hierarchy, and class collaborations will be invaluable when you are required to assess OO design quality. A class encapsulates data and the function that manipulate the data. It is often the "parent" for subclasses (sometimes called children) that inherit its attributes and operations. It often collaborates with other classes. Each of these characteristics can be used as the basis for measurement.[7]

Chidamber and Kemerer (CK) have proposed one of the most widely referenced sets of OO software metrics [Chi94]. Often referred to as the *CK metrics suite,* the authors have proposed six class-based design metrics for OO systems.[8]

Weighted methods per class (WMC). Assume that n methods of complexity c_1, c_2, . . . , c_n are defined for a class **C.** The specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on a value of 1.0.

$$\text{WMC} = \Sigma\, c_i$$

for $i = 1$ to n. The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class. In addition, the larger the number of methods, the more complex is the inheritance tree (all subclasses inherit the methods of their parents). Finally, as the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse. For all of these reasons, WMC should be kept as low as is reasonable.

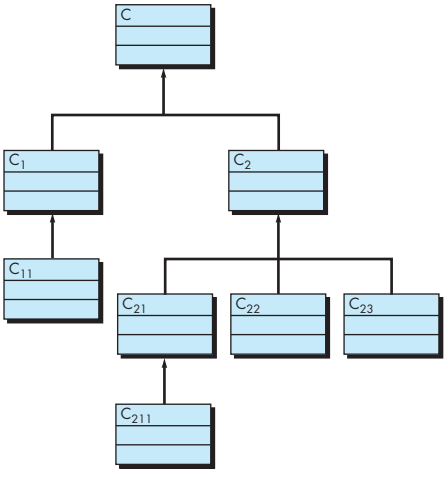
Depth of the inheritance tree (DIT). This metric is "the maximum length from the node to the root of the tree" [Chi94]. Referring to Figure 30.5, the value of DIT for the class hierarchy shown is 4. As DIT grows, it is likely that lower-level classes will inherit many methods. This leads to potential difficulties when attempting to predict the behavior of a class. A deep class hierarchy (DIT is large)

7 It should be noted that the validity of some of the metrics discussed in this chapter is currently debated in the technical literature. Those who champion measurement theory demand a degree of formalism that some OO metrics do not provide. However, it is reasonable to state that the metrics noted provide useful insight for the software engineer.

8 Chidamber, Darcy, and Kemerer use the term *methods* rather than *operations*. Their usage of the term is reflected in this section.

FIGURE 30.5

A class hierarchy



also leads to greater design complexity. On the positive side, large DIT values imply that many methods may be reused.

ADVICE

The concepts of coupling and cohesion apply to both conventional and OO software. Keep class coupling low and class and operation cohesion high.

Number of children (NOC). The subclasses that are immediately subordinate to a class in the class hierarchy are termed its children. Referring to Figure 30.5, class $\mathbf{C}_2$ has three children—subclasses $\mathbf{C}_{21}$, $\mathbf{C}_{22}$, and $\mathbf{C}_{30}$. As the number of children grows, reuse increases, but also, as NOC increases, the abstraction represented by the parent class can be diluted if some of the children are not appropriate members of the parent class. As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

Coupling between object classes (CBO). The CRC model (Chapter 10) may be used to determine the value for CBO. In essence, CBO is the number of collaborations listed for a class on its CRC index card.[9] As CBO increases, it is likely that the reusability of a class will decrease. High values of CBO also complicate modifications and the testing that ensues when modifications are made. In general, the CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.

Response for a class (RFC). The response set of a class is "a set of methods that can potentially be executed in response to a message received by an object of that class" [Chi94]. RFC is the number of methods in the response set. As RFC increases, the effort required for testing also increases because the test sequence

9 If CRC index cards are developed manually, completeness and consistency must be assessed before CBO can be determined reliably.

(Chapter 24) grows. It also follows that, as RFC increases, the overall design complexity of the class increases.

Lack of cohesion in methods (LCOM). Each method within a class **C** accesses one or more attributes (also called instance variables). LCOM is the number of methods that access one or more of the same attributes.[10] If no methods access the same attributes, then LCOM = 0. To illustrate the case where LCOM ≠ 0, consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes). Therefore, LCOM = 4. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high; that is, keep LCOM low.[11]

SafeHome

Applying CK Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, Shakira, and Ed—members of the *SafeHome* software engineering team who are continuing to work on component-level design and test-case design.

The conversation:

Vinod: Did you guys get a chance to read the description of the CK metrics suite I sent you on Wednesday and make those measurements?

Shakira: Wasn't too complicated. I went back to my UML class and sequence diagrams, like you suggested, and got rough counts for DIT, RFC, and LCOM. I couldn't find the CRC model, so I didn't count CBO.

Jamie (smiling): You couldn't find the CRC model because I had it.

Shakira: That's what I love about this team, superb communication.

Vinod: I did my counts . . . did you guys develop numbers for the CK metrics?

[Jamie and Ed nod in the affirmative.]

Jamie: Since I had the CRC cards, I took a look at CBO and it looked pretty uniform across most of the classes. There was one exception, which I noted.

Ed: There are a few classes where RFC is pretty high, compared with the averages . . . maybe we should take a look at simplifying them.

Jamie: Maybe yes, maybe no. I'm still concerned about time, and I don't want to fix stuff that isn't really broken.

Vinod: I agree with that. Maybe we should look for classes that have bad numbers in at least two or more of the CK metrics. Kind of two strikes and you're modified.

Shakira (looking over Ed's list of classes with high RFC): Look, see this class, it's got a high LCOM as well as a high RFC. Two strikes?

Vinod: Yeah I think so . . . it'll be difficult to implement because of complexity and difficult to test for the same reason. Probably worth designing two separate classes to achieve the same behavior.

Jamie: You think modifying it'll save us time?

Vinod: Over the long haul, yes.

10 The formal definition is a bit more complex. See [Chi94] for details.

11 The LCOM metric provides useful insight in some situations, but it can be misleading in others. For example, keeping coupling encapsulated within a class increases the cohesion of the system as a whole. Therefore in at least one important sense, higher LCOM actually suggests that a class may have higher cohesion, not lower.

30.3.4 Class-Oriented Metrics—The MOOD Metrics Suite

Quote:

"Analyzing OO software in order to evaluate its quality is becoming increasingly important as the [OO] paradigm continues to increase in popularity."

Rachel Harrison et al.

Harrison, Counsell, and Nithi [Har98b] propose a set of metrics for object-oriented design that provides quantitative indicators for OO design characteristics. A sampling of MOOD metrics follows.

Method inheritance factor (MIF). The degree to which the class architecture of an OO system makes use of inheritance for both methods (operations) and attributes is defined as

$$\text{MIF} = \frac{\Sigma M_i(C_i)}{\Sigma M_a(C_i)}$$

where the summation occurs over $i = 1$ to TC. TC is defined as the total number of classes in the architecture, C_i is a class within the architecture, and

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

where

$M_a(C_i)$ = number of methods that can be invoked in association with C_i

$M_d(C_i)$ = number of methods declared in the class C_i

$M_i(C_i)$ = number of methods inherited (and not overridden) in C_i

The value of MIF [the attribute inheritance factor (AIF) is defined in an analogous manner] provides an indication of the impact of inheritance on the OO software.

Coupling factor (CF). Earlier in this chapter I noted that coupling is an indication of the connections between elements of the OO design. The MOOD metrics suite defines coupling in the following way:

$$\text{CF} = \Sigma_i \Sigma_j \, is_client \frac{(C_i, C_j)}{\text{TC}^2 - \text{TC}}$$

where the summations occur over $i = 1$ to TC and $j = 1$ to TC. The function

$is\ client$ = 1, if and only if a relationship exists between the client class C_c and the server class C_s, and $C_c \neq C_s$

= 0, otherwise

Although many factors affect software complexity, understandability, and maintainability, it is reasonable to conclude that as the value for CF increases, the complexity of the OO software will also increase and understandability, maintainability, and the potential for reuse may suffer as a result.

Harrison and her colleagues [Har98b] present a detailed analysis of MIF and CF along with other metrics and examine their validity for use in the assessment of design quality.

30.3.5 OO Metrics Proposed by Lorenz and Kidd

In their book on OO metrics, Lorenz and Kidd [Lor94] divide class-based metrics into four broad categories that each have a bearing on component-level design: size, inheritance, internals, and externals. Size-oriented metrics for an OO design class focus on counts of attributes and operations for an individual class and average values for the OO system as a whole. Inheritance-based metrics focus on the manner in which operations are reused through the class hierarchy. Metrics for class internals look at cohesion (Section 30.3.3) and code-oriented issues, and external metrics examine coupling and reuse. One example of the metrics proposed by Lorenz and Kidd is as follows:

ADVICE

During review of the analysis model, CRC index cards will provide a reasonable indication of expected values for CS. If you encounter a class with a large number of responsibilities, consider partitioning it.

Class size (CS). The overall size of a class can be determined using the following measures:

- The total number of operations (both inherited and private instance operations) that are encapsulated within the class
- The number of attributes (both inherited and private instance attributes) that are encapsulated by the class

The WMC metric proposed by Chidamber and Kemerer (Section 30.3.3) is also a weighted measure of class size. As we noted earlier, large values for CS indicate that a class may have too much responsibility. This will reduce the reusability of the class and complicate implementation and testing. In general, inherited or public operations and attributes should be weighted more heavily in determining class size [Lor94]. Private operations and attributes enable specialization and are more localized in the design. Averages for the number of class attributes and operations may also be computed. The lower the average values for size, the more likely that classes within the system can be reused widely.

KEY POINT

It is possible to compute measures of the functional independence—coupling and cohesion—of a component and to use these to assess the quality of a design.

30.3.6 Component-Level Design Metrics

Component-level design metrics for conventional software components focus on internal characteristics of a software component and include measures of the "three Cs"—module cohesion [Bie94], coupling [Dha95], and complexity [Zus97]. These measures can help you judge the quality of a component-level design.

Component-level design metrics may be applied once a procedural design has been developed and are "glass box" in the sense that they require knowledge of the inner workings of the module under consideration. Alternatively, they may be delayed until source code is available.

30.3.7 Operation-Oriented Metrics

Because the class is the dominant unit in OO systems, fewer metrics have been proposed for operations that reside within a class. Churcher and Shepperd

[Chu95] discuss this when they state: "Results of recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [Wil93], suggesting that connectivity structure of a system may be more important than the content of individual modules." However, some insights can be gained by examining average characteristics for methods (operations). Three simple metrics, proposed by Lorenz and Kidd [Lor94], are appropriate:

Average operation size (OS_{avg}). Size can be determined by counting the number of lines of code or the number of messages sent by the operation. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well allocated within a class.

Operation complexity (OC). The complexity of an operation can be computed using any of the complexity metrics proposed for conventional software [Zus90]. Because operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.

Average number of parameters per operation (NP_{avg}). The larger the number of operation parameters, the more complex the collaboration between objects. In general, NP_{avg} should be kept as low as possible.

30.3.8 User Interface Design Metrics

Although there is significant literature on the design of human/computer interfaces (Chapter 15), relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

Quote:

"You can learn at least one principle of user interface design by loading a dishwasher. If you crowd a lot in there, nothing gets very clean."

Author unknown

A study of Web page metrics [Ivo01] indicates that simple characteristics of the elements of the layout can also have a significant impact on the perceived quality of the GUI design. The number of words, links, graphics, colors, and fonts (among other characteristics) contained within a Web page affect the perceived complexity and quality of that page.

Although UI metrics may be useful in some cases, the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [Nie94] report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated."

30.4 Design Metrics for Web and Mobile Apps

A useful set of measures and metrics for WebApps provides quantitative answers to the following questions:

- Does the user interface promote usability?
- Are the aesthetics of the WebApp appropriate for the application domain and pleasing to the user?

- Is the content designed in a manner that imparts the most information with the least effort?
- Is navigation efficient and straightforward?
- Has the WebApp architecture been designed to accommodate the special goals and objectives of WebApp users, the structure of content and functionality, and the flow of navigation required to use the system effectively?
- Are components designed in a manner that reduces procedural complexity and enhances the correctness, reliability, and performance?

Today, each of these questions can be addressed only qualitatively because a validated suite of metrics that would provide quantitative answers does not yet exist.

In the paragraphs that follow, we present a representative sampling of Web and MobileApp design metrics that have been proposed in the literature. It is important to note that many of these metrics have not as yet been validated and should be used judiciously.

Interface metrics. For WebApps, the following interface measures can be considered:

Suggested Metric	*Description*
Layout appropriateness	The relative position of entities within the interface
Layout complexity	Number of distinct regions[12] defined for an interface
Layout region complexity	Average number of distinct links per region
Recognition complexity	Average number of distinct items the user must look at before making a navigation or data input decision
Recognition time	Average time (in seconds) that it takes a user to select the appropriate action for a given task
Typing effort	Average number of key strokes required for a specific function
Mouse pick effort	Average number of mouse picks per function
Selection complexity	Average number of links that can be selected per page
Content acquisition time	Average number of words of text per Web page
Memory load	Average number of distinct data items that the user must remember to achieve a specific objective

ADVICE

Many of these metrics are applicable to all user interfaces and should be considered in conjunction with those presented in Section 30.3.8.

Aesthetic (graphic design) metrics. By its nature, aesthetic design relies on qualitative judgment and is not generally amenable to measurement and metrics.

12 A distinct region is an area within the layout display that accomplishes some specific set of related functions (e.g., a menu bar, a static graphical display, a content area, an animated display).

However, Ivory and her colleagues [Ivo01] propose a set of measures that may be useful in assessing the impact of aesthetic design:

Suggested Metric	*Description*
Word count	Total number of words that appear on a page
Body text percentage	Percentage of words that are body versus display text (e.g., headers)
Emphasized body text percentage	Portion of body text that is emphasized (e.g., bold, capitalized)
Text positioning count	Changes in text position from flush left
Text cluster count	Text areas highlighted with color, bordered regions, rules, or lists
Link count	Total links on a page
Page size	Total bytes for the page as well as elements, graphics, and style sheets
Graphic percentage	Percentage of page bytes that are for graphics
Graphics count	Total graphics on a page (not including graphics specified in scripts, applets, and objects)
Color count	Total colors employed
Font count	Total fonts employed (i.e., face + size + bold + italic)

Content metrics. Metrics in this category focus on content complexity and on clusters of content objects that are organized into pages [Men01].

Suggested Metric	*Description*
Page wait	Average time required for a page to download at different connection speeds
Page complexity	Average number of different types of media used on page, not including text
Graphic complexity	Average number of graphics media per page
Audio complexity	Average number of audio media per page
Video complexity	Average number of video media per page
Animation complexity	Average number of animations per page
Scanned image complexity	Average number of scanned images per page

Navigation metrics. Metrics in this category address the complexity of the navigational flow [Men01]. In general, they are applicable only for static Web applications, which don't include dynamically generated links and pages.

Suggested Metric	*Description*
Page-linking complexity	Number of links per page
Connectivity	Total number of internal links, not including dynamically generated links
Connectivity density	Connectivity divided by page count

Using a subset of the metrics suggested, it may be possible to derive empirical relations that allow a WebApp development team to assess technical quality and predict effort based on projected estimates of complexity. Further work remains to be accomplished in this area.

SOFTWARE TOOLS

Technical Metrics for WebApps

Objective: To assist Web engineers in developing meaningful WebApp metrics that provide insight into the overall quality of an application.

Mechanics: Tool mechanics vary.

Representative Tools:[13]

Netmechanic Tools, developed by Netmechanic (**www.netmechanic.com**), is a collection of tools that help to improve website performance, focusing on implementation-specific issues.

NIST Web Metrics Testbed, developed by the National Institute of Standards and Technology (**zing.ncsl.nist.gov/WebTools/**) encompasses the following collection of useful tools that are available for download:

Web Static Analyzer Tool (WebSAT)—Checks Web page HTML against typical usability guidelines.

Web Category Analysis Tool (WebCAT)—Lets the usability engineer construct and conduct a Web category analysis.

Web Variable Instrumenter Program (WebVIP)—Provides instruments to a website to capture a log of user interaction.

Framework for Logging Usability Data (FLUD)—Implements a file formatter and parser for representation of user interaction logs.

VisVIP Tool—Produces a 3D visualization of user navigation paths through a website.

TreeDec—Adds navigation aids to the pages of a website.

30.5 METRICS FOR SOURCE CODE

Halstead's theory of "software science" [Hal77] proposed the first analytical "laws" for computer software.[14] Halstead assigned quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. The measures are:

Quote:

"The human brain follows a more rigid set of rules [for developing algorithms] than it has been aware of."

Maurice Halstead

n_1 = number of distinct operators that appear in a program
n_2 = number of distinct operands that appear in a program
N_1 = total number of operator occurrences
N_2 = total number of operand occurrences

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and

13 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category.

14 It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed (e.g., [Fel89]).

ADVICE

Operators include all flow of control constructs, conditionals, and math operations. Operands encompass all program variables and constants.

other features such as development effort, development time, and even the projected number of faults in the software.

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2)$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

Halstead's work is amenable to experimental verification and a large body of research has been conducted to investigate software science. A discussion of this work is beyond the scope of this book. For further information, see [Zus90], [Fen91], and [Zus97].

30.6 Metrics for Testing

The majority of metrics proposed for testing focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

Architectural design metrics provide information on the ease or difficulty associated with integration testing (Section 30.3) and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing, a test-case design method presented in Chapter 23. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, you should expend above average effort to uncover errors in such modules before they are integrated in a system.

KEY POINT

Testing metrics fall into two broad categories: (1) metrics that attempt to predict the likely number of tests required at various testing levels, and (2) metrics that focus on test coverage for a given component.

30.6.1 Halstead Metrics Applied to Testing

Testing effort can be estimated using metrics derived from Halstead measures (Section 30.5). Using the definitions for program volume V and program level PL, Halstead effort e can be computed as

$$PL = \frac{1}{[(n_1/2) \times (N_2/n_2)]} \tag{30.6a}$$

$$e = \frac{V}{PL} \tag{30.6b}$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\text{Percentage of testing effort } (k) = \frac{e(k)}{\Sigma e(i)} \tag{30.7}$$

where $e(k)$ is computed for module k using Equations (30.6) and the summation in the denominator of Equation (30.7) is the sum of Halstead effort across all modules of the system.

30.6.2 Metrics for Object-Oriented Testing

The OO design metrics noted in Section 30.3 provide an indication of design quality. They also provide a general indication of the amount of testing effort required to exercise an OO system. Binder [Bin94b] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system. The metrics consider aspects of encapsulation and inheritance.

Lack of cohesion in methods (LCOM).[15] The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

Percent public and protected (PAP). Public attributes are inherited from other classes and therefore are visible to those classes. Protected attributes are accessible to methods in subclasses. This metric indicates the percentage of class attributes that are public or protected. High values for PAP increase the likelihood of side effects among classes because public and protected attributes lead to high potential for coupling.[16] Tests must be designed to ensure that such side effects are uncovered.

Public access to data members (PAD). This metric indicates the number of classes (or methods) that can access another class's attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

ADVICE

OO testing can be quite complex. Metrics can assist you in targeting testing resources at threads, scenarios, and packages of classes that are "suspect" based on measured characteristics. Use them.

Number of root classes (NOR). This metric is a count of the distinct class hierarchies that are described in the design model. Test suites for each root class and the corresponding class hierarchy must be developed. As NOR increases, testing effort also increases.

Fan-in (FIN). When used in the OO context, fan-in in the inheritance hierarchy is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

15 See Section 30.3.3 for a description of LCOM.

16 Some people promote designs with none of the attributes being public or private, that is, PAP = 0. This implies that all attributes must be accessed in other classes via methods.

Number of children (NOC) and depth of the inheritance tree (DIT).[17] As we mentioned in Chapter 24, superclass methods will have to be retested for each subclass.

30.7 Metrics for Maintenance

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-2005 [IEE05] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_T = number of modules in the current release
F_c = number of modules in the current release that have been changed
F_a = number of modules in the current release that have been added
F_d = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{[M_T - (F_a + F_c + F_d)]}{M_T}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

Software Tools

Product Metrics

Objective: To assist software engineers in developing meaningful metrics that assess the work products produced during analysis and design modeling, source code generation, and testing.

Mechanics: Tools in this category span a broad array of metrics and are implemented either as a stand-alone application or (more commonly) as functionality that exists within tools for analysis and design, coding, or testing. In most cases, the metrics tool analyzes a representation of the software (e.g., a UML model or source code) and develops one or more metrics as a result.

Representative Tools:[18]

Krakatau Metrics, developed by Power Software (**www.powersoftware.com/products**), computes complexity, Halstead, and related metrics for C/C++ and Java.

Rational Rose, distributed by IBM (**http://www-01.ibm.com/software/awdtools/developer/rose/**), is a comprehensive tool set for UML modeling that incorporates a number of metrics analysis features.

RSM, developed by M-Squared Technologies (**http://msquaredtechnologies.com/Resource-Standard-Metrics.html**), computes a wide variety of code-oriented metrics for C, C++, and Java.

Understand, developed by Scientific Toolworks, Inc. (**www.scitools.com**), calculates code-oriented metrics for a variety of programming languages.

17 See Section 30.3.3 for a description of NOC and DIT.

18 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category.

30.8 Summary

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling you to assess quality before the product is built. Metrics provide the insight necessary to create effective requirements and design models, solid code, and thorough tests.

To be useful in a real-world context, a software metric must be simple and computable, persuasive, consistent, and objective. It should be programming language independent and provide you with effective feedback.

Metrics for the requirements model focus on function, data, and behavior—the three components of the model. Metrics for design consider architecture, component-level design, and interface design issues. Architectural design metrics consider the structural aspects of the design model. Component-level design metrics provide an indication of module quality by establishing indirect measures for cohesion, coupling, and complexity. User interface design metrics provide an indication of the ease with which a GUI can be used. WebApp metrics consider aspects of the user interface as well as WebApp aesthetics, content, and navigation.

Metrics for OO systems focus on measurement that can be applied to the class and the design characteristics—localization, encapsulation, information hiding, inheritance, and object abstraction techniques—that make the class unique. The CK metrics suite defines six class-oriented software metrics that focus on the class and the class hierarchy. The metrics suite also develops metrics to assess the collaborations between classes and the cohesion of methods that reside within a class. At a class-oriented level, the CK metrics suite can be augmented with metrics proposed by Lorenz and Kidd and the MOOD metrics suite.

Halstead provides an intriguing set of metrics at the source code level. Using the number of operators and operands present in the code, software science provides a variety of metrics that can be used to assess program quality.

Few product metrics have been proposed for direct use in software testing and maintenance. However, many other product metrics can be used to guide the testing process and as a mechanism for assessing the maintainability of a computer program. A wide variety of OO metrics have been proposed to assess the testability of an OO system.

Problems and Points to Ponder

30.1. Measurement theory is an advanced topic that has a strong bearing on software metrics. Using [Zus97], [Fen91], [Zus90] or Web-based sources, write a brief paper that outlines the main tenets of measurement theory. Individual project: Develop a presentation on the subject and present it to your class.

30.2. Why is it that a single, all-encompassing metric cannot be developed for program complexity or program quality? Try to come up with a measure or metric from everyday life that violates the attributes of effective software metrics defined in Section 30.1.5.

30.3. A system has 12 external inputs, 24 external outputs, fields 30 different external queries, manages 4 internal logical files, and interfaces with 6 different legacy systems (6 EIFs). All of these data are of average complexity and the overall system is relatively simple. Compute FP for the system.

30.4. Software for System X has 24 individual functional requirements and 14 nonfunctional requirements. What is the specificity of the requirements? The completeness?

30.5. A major information system has 1140 modules. There are 96 modules that perform control and coordination functions and 490 modules whose function depends on prior processing. The system processes approximately 220 data objects that each have an average of three attributes. There are 140 unique database items and 90 different database segments. Finally, 600 modules have single entry and exit points. Compute the DSQI for this system.

30.6. A class **X** has 12 operations. Cyclomatic complexity has been computed for all operations in the OO system, and the average value of module complexity is 4. For class **X**, the complexity for operations 1 to 12 is 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectively. Compute the weighted methods per class.

30.7. Develop a software tool that will compute cyclomatic complexity for a programming language module. You may choose the language.

30.8. Develop a small software tool that will perform a Halstead analysis on programming language source code of your choosing.

30.9. A legacy system has 940 modules. The latest release required that 90 of these modules be changed. In addition, 40 new modules were added and 12 old modules were removed. Compute the software maturity index for the system.

Further Readings and Information Sources

There is a surprisingly large number of books dedicated to software metrics, although the majority focus on process and project metrics to the exclusion of product metrics. Jones and Bonsignour (*The Economics of Software Quality,* Addison-Wesley, 2011), Lanza and her colleagues (*Object-Oriented Metrics in Practice,* Springer, 2010) and Jones (*Applied Software Measurement: Global Analysis of Productivity and Quality,* McGraw-Hill, 2008) discuss OO metrics and their use for assessing the quality of a design. Genero (*Metrics for Software Conceptual Models,* Imperial College Press, 2005) and Ejiogu (*Software Metrics,* BookSurge Publishing, 2005) present a wide variety of technical metrics for use cases, UML models, and other modeling representations. Hutcheson (*Software Testing Fundamentals: Methods and Metrics,* Wiley, 2003) presents a set of metrics for testing. Abran (*Software Metrics and Software Metrology,* Wiley-IEEE Computer Society, 2010), Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2nd ed., 2002), Fenton and Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Brooks-Cole Publishing, 1998), and Zuse [Zus97] have written thorough treatments of product metrics.

Books by Card and Glass [Car90], Zuse [Zus90], Fenton [Fen91], Ejiogu [Eji91], Moeller and Paulish (*Software Metrics,* Chapman and Hall, 1993), and Hetzel [Het93] all address product metrics in some detail. Oman and Pfleeger (*Applying Software Metrics,* IEEE Computer Society Press, 1997) have edited an anthology of important papers on software metrics.

The International Function Point Users group has published a book on using function metrics (*The IFPUG Guide or OT and Software Measurement,* Auerbach Publications, 2012). Methods for establishing a metrics program and the underlying principles for software measurement are considered by Ebert and his colleagues (*Best Practices in Software Measurement,* Springer, 2004). Shepperd (*Foundations of Software Measurement,* Prentice Hall, 1996) also addresses measurement theory in some detail. Current research is presented in the *Proceedings of the Symposium on Software Metrics* (IEEE, published annually).

A comprehensive summary of dozens of useful software metrics is presented in [IEE93]. In general, a discussion of each metric has been distilled to the essential "primitives" (measures) required to compute the metric and the appropriate relationships to effect the computation. An appendix provides discussion and many references.

Whitmire [Whi97] presents a comprehensive and mathematically sophisticated treatment of OO metrics. Lorenz and Kidd [Lor94] and Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity,* Prentice Hall, 1996) provide treatments that are dedicated to OO metrics.

A wide variety of information sources on software metrics are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software metrics can be found at the SEPA website: **www.mhhe.com/pressman**.

This page intentionally left blank

PART

Four

Managing Software Projects

In this part of *Software Engineering: A Practitioner's Approach* you'll learn the management techniques required to plan, organize, monitor, and control software projects. These questions are addressed in the chapters that follow:

- How must people, process, and problem be managed during a software project?
- How can software metrics be used to manage a software project and the software process?
- How does a software team generate reliable estimates of effort, cost, and project duration?
- What techniques can be used to formally assess the risks that can have an impact on project success?
- How does a software project manager select the set of software engineering work tasks?
- How is a project schedule created?
- Why are maintenance and reengineering so important for both software engineering managers and practitioners?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product.

CHAPTER

31 Project Management Concepts

Key Concepts

agile teams. 691
coordination and communication . . . 692
critical practices . . 699
people 687
problem decomposition. . . . 694
product. 693
project 697
software scope. . . 694
software team . . . 689
stakeholders. 687
team leaders. 688
W^{5}HH principle . . . 698

In the preface to his book on software project management, Meiler Page-Jones [Pag85] makes a statement that can be echoed by many software engineering consultants:

> I've visited dozens of commercial shops, both good and bad, and I've observed scores of [IT] managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be

Quick Look

What is it? Although many of us (in our darker moments) take Dilbert's view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to full operational deployment.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies among people involved in a software project. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software professionals.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four Ps—people, product, process, and project. People must be organized to perform software work effectively. Communication with the customer and other stakeholders must occur so that product scope and requirements are understood. A process that is appropriate for the people and the product should be selected. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and identifying mechanisms to monitor and control work defined by the plan.

What is the work product? A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until you've delivered a high-quality product on time and within budget. However, a project manager does it right when he encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

conducted for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and Chapters 32 through 37, we'll present the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 32 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost are discussed in Chapter 33. Chapter 34 will help you to define a realistic project schedule. The management activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 35. Chapter 36 considers maintenance and reengineering and discusses the management issues that you'll encounter when you must deal with legacy systems. Finally, Chapter 37 discussed techniques for studying and improving your team's software engineering processes.

31.1 The Management Spectrum

Effective software project management focuses on the four Ps: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

31.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives" [Cur01].

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, and team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

The People-CMM is a companion to the *Software Capability Maturity Model–Integration* (Chapter 37) that guides organizations in the creation of a mature

software process. Issues associated with people management and structure for software projects are considered later in this chapter.

31.1.2 The Product

ADVICE

Those who adhere to the agile process philosophy (Chapter 5) argue that their process is leaner than others. That may be true, but they still have a process, and agile software engineering still requires discipline.

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering (Chapter 8). Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

31.1.3 The Process

Quote:

"A project is like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. But most projects worth doing are more like driving a truck off-road, in the mountains, at night."

Cem Kaner, James Bach, and Bret Pettichord

A software process (Chapters 3–5) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

31.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below

35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.[1]

To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 31.5 and in the chapters that follow.

31.2 People

People build computer software, and projects succeed because well-trained, motivated people get things done. All of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue that people are primary, but their actions sometimes belie their words. In this section we examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering.

31.2.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. *Senior managers* who define the business issues that often have a significant influence on the project.
2. *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
3. *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
4. *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. *End users* who interact with the software once it is released for production use.

1 Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, we think, is that a substantial number of these "failed" projects are ill conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

Every software project is populated by people who fall within this taxonomy.[2] To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

31.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers" [Edg95].

? What do we look for when choosing someone to lead a software project?

In an excellent book of technical leadership, Jerry Weinberg [Wei86] suggests an MOI model of leadership:

> **Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.
>
> **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
>
> **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem-solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Quote:

"In simplest terms, a leader is one who knows where he wants to go, and gets up, and goes."

John Erskine

Another view [Edg95] of the characteristics that define an effective project manager emphasizes four key traits:

> **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.
>
> **Managerial identity.** A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.
>
> **Achievement.** A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. She must

2 When Web or mobile apps are developed, other nontechnical people may be involved in content creation.

demonstrate through her own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

31.2.3 The Software Team

Quote:

"Not every group is a team, and not every team is effective."

Glenn Parker

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

? What factors should be considered when the structure of a software team is chosen?

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes seven project factors that should be considered when planning the structure of software engineering teams: (1) difficulty of the problem to be solved; (2) "size" of the resultant program(s) in lines of code or function points; (3) time that the team will stay together (team lifetime); (4) degree to which the problem can be modularized; (5) quality and reliability of the system to be built; (6) rigidity of the delivery date, and (7) degree of sociability (communication) required for the project.

Constantine [Con93] suggests four "organizational paradigms" for software engineering teams:

? What options do we have when defining the structure of a software team?

1. A *closed paradigm* structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. A *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. An *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.

4. A *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

Quote:

"If you want to be incrementally better: Be competitive. If you want to be exponentially better: Be cooperative."

Author unknown

As an historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the *chief programmer team.* This structure was first proposed by Harlan Mills and described by Baker [Bak72]. As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed *innovative anarchy.* Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team: team members must have trust in one another, the distribution of skills must be appropriate to the problem, and mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book *Peopleware,* DeMarco and Lister [DeM98] discuss this issue:

? What is a "jelled" team?

> We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

? Why is it that teams fail to jell?

But not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless

Quote:

"Do or do not. There is no try."

Yoda from *Star Wars*

absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built, the people doing the work, and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[3] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverts. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last-minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.[4] However, it is important to note that recognition of human differences is the first step toward creating teams that jell.

31.2.4 Agile Teams

Many software organizations advocate agile software development (Chapter 5) as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity.

The small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled

3 Technical reviews are discussed in detail in Chapter 20.

4 An excellent introduction to these issues as they relate to software project teams can be found in [Fer98].

with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write:

> If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job . . .

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing*. A self-organizing team does not necessarily maintain a single team structure but instead, uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 31.2.3.

KEY POINT

An agile team is a self-organizing team that has autonomy to plan and make technical decisions.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

31.2.5 Coordination and Communication Issues

Quote:

"Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the [agile] team, not individuals who make up the team."

Jim Highsmith

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively

non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

SafeHome

Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold on that for a moment. I'd like to talk about how we are going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and that fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): "We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

Doug: And, we're going to use UML as our modeling approach.

Vinod: But keep extraneous documentation to an absolute minimum.

Doug: Who is the liaison with me?

Jamie: We decided that Vinod will be the tech lead—he's got the most experience, so Vinod is your liaison, but feel free to talk to any of us.

Doug (laughing): Don't worry, I will.

31.3 The Product

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide information necessary for estimates, but analysis often takes weeks or even months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed now!

Like it or not, you must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

ADVICE

If you can't bound a characteristic of the software you intend to build, list the characteristic as a project risk (Chapter 35).

31.3.1 Software Scope

The first software project management activity is the determination of *software scope*. Scope is defined by answering the following questions:

Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described.

31.3.2 Problem Decomposition

ADVICE

In order to develop a reasonable project plan, you must decompose the problem. This can be accomplished using a list of functions or with use cases or for agile work, user stories.

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapters 8–11). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality and content (information) that must be delivered and (2) the process that will be used to deliver it.

Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 33). Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software.

31.4 The Process

The framework activities (Chapter 2) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by your project team.

Your team must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities, must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 33.

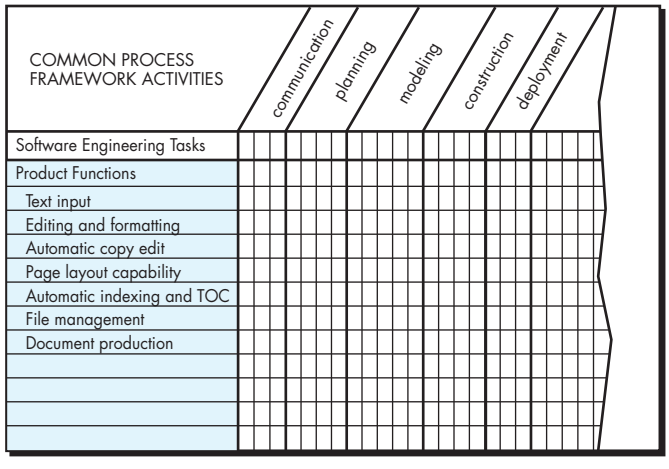
31.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.

Assume that the organization has adopted the generic framework activities—**communication, planning, modeling, construction,** and **deployment**—discussed in Chapter 2. The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 31.1 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.[5] The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the

FIGURE 31.1

Melding the problem and the process



5 It should be noted that work tasks must be adapted to the specific needs of the project based on a number of adaptation criteria.

KEY POINT

The process framework establishes a skeleton for project planning. It is adapted by allocating a task set that is appropriate to the project.

tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 26.

31.4.2 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.[6]

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?" For example, a small, relatively simple project might require the following work tasks for the communication activity:

1. Develop list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the **communication**:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.

6 Recall that project characteristics also have a strong bearing on the structure of the software team (Section 31.2.3).

5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software. Alternatively, develop use cases that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use cases with all concerned.
10. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call **communication**, but the first project team performs half as many software engineering work tasks as the second.

31.5 The Project

? What are the signs that a software project is in jeopardy?

In order to manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines signs that indicate that an information systems project is in jeopardy. In some cases, software people don't understand their customer's needs. This leads to a project with a poorly defined scope. In other projects, changes are managed poorly. Sometimes, the chosen technology changes or business needs shift and management sponsorship is lost. Management can set unrealistic deadlines or end users can be resistant to the new system. There are cases in which the project team simply does not have the requisite skills. And finally, there are developers who never seem to learn from their mistakes.

Jaded industry professionals often refer to the "90–90 rule" when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time [Zah94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [Ree99] suggests a five-part commonsense approach to software projects:

1. *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 31.2.3) and giving the team the autonomy, authority, and technology needed to do the job.
2. *Maintain momentum.* Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide

incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.[7]

3. *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 32) can be collected and used to assess progress against averages developed for the software development organization.
4. *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).
5. *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

31.6 The W^{5}HH Principle

In an excellent paper on software process and projects, Barry Boehm [Boe96] states: "[Y]ou need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the *W^{5}HH Principle,* after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

? How do we define key project characteristics?

Why is the system being developed? All stakeholders should assess the validity of business reasons for the software work. Does the business purpose justify the expenditure of people, time, and money?

What will be done? The task set required for the project is defined.

When will it be done? The team establishes a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

7 The implication of this statement is that bureaucracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is deemphasized. The team should be self-organizing and autonomous.

Who is responsible for a function? The role and responsibility of each member of the software team is defined.

Where are they located organizationally? Not all roles and responsibilities reside within software practitioners. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 33) based on answers to earlier questions.

Boehm's W^5HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide you and your team with an excellent planning outline.

31.7 Critical Practices

The Airlie Council[8] has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [Air99].

Critical practices[9] include: metric-based project management (Chapter 32), empirical cost and schedule estimation (Chapters 33 and 34), earned value tracking (Chapter 34), defect tracking against quality targets (Chapters 19 though 21), and people aware management (Section 31.2). Each of these critical practices is addressed throughout Part 4 of this book.

Software Tools

Software Tools for Project Managers

The tools listed here are generic and apply to a broad range of activities performed by project managers. Specific project management tools (e.g., scheduling tools, estimating tools, risk analysis tools) are considered in later chapters.

Representative Tools:[10]

Projectmanagement.com (**http://www.projectmanagement.com**) has developed a set of useful *checklists for project managers*.

Ittoolkit.com (**www.ittoolkit.com**) provides "a collection of planning guides, process templates and smart worksheets" available on CD-ROM.

8 The Airlie Council was comprised of a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering. For more on best practices, see **http://www.swqual.com/e_newsletter.html**.

9 Only those critical practices associated with "project integrity" are noted here.

10 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

31.8 Summary

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the modeling, construction, and deployment of computer software.

Four Ps have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. Product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, technical reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. Each of these topics is considered in the chapters that follow.

Problems and Points to Ponder

31.1. Based on information contained in this chapter and your own experience, develop "10 commandments" for empowering software engineers. That is, make a list of 10 guidelines that will lead to software people who work to their full potential.

31.2. The Software Engineering Institute's People Capability Maturity Model (People-CMM) takes an organized look at "key practice areas" (KPAs) that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

31.3. Describe three real-life situations in which the customer and the end user are the same. Describe three situations in which they are different.

31.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

31.5. Review a copy of Weinberg's book [Wei86], and write a two- or three-page summary of the issues that should be considered in applying the MOI model.

31.6. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

31.7. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team structure would you choose and why? What software process model(s) would you choose and why?

31.8. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next-generation version of its widely used word-processing software. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

31.9. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal is to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

31.10. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

31.11. Do a first-level functional decomposition of the page layout function discussed briefly in Section 31.3.2.

Further Readings and Information Sources

The Project Management Institute (*Guide to the Project Management Body of Knowledge,* 4th ed., PMI, 2009) covers all important aspects of project management. Wysocki (*Effective Software Project Management: Traditional, Agile, Extreme,* 6th ed., Wiley, 2011), Slinger and Broderick (*The Software Project Manager's Bridge to Agility,* Addison-Wesley, 2008), Bechtold (*Essentials of Software Project Management,* 2nd ed., Management Concepts, 2007), Stellman and Greene (*Applied Software Project Management,* O'Reilly, 2005), and Berkun (*Making Things Happen: Mastering Project Management—Theory in Practice,* O'Reilly, 2008) teach basic skills and provide detailed guidance for all software project management tasks. McConnell (*Professional Software Development,* Addison-Wesley, 2004) offers pragmatic advice for achieving "shorter schedules, higher quality products, and more successful projects." Henry (*Software Project Management,* Addison-Wesley, 2003) offers real-world advice that is useful for all project managers.

Tom DeMarco and his colleagues (*Adrenaline Junkies and Template Zombies,* Dorset House, 2008) have written an insightful treatment of the human patterns that are encountered in every software project. An excellent four-volume series written by Weinberg (*Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts, explains how to use measurements effectively, and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Futrell and his colleagues (*Quality Software Project Management,* Prentice Hall, 2002) present a voluminous treatment of project management. Books by Neill and his colleagues (*Antipatterns: Managing Software Organizations,* 2nd ed., Auerbach Publications, 2011) and Brown and his colleagues (*Antipatterns in Project Management,* Wiley, 2000) discuss what not to do during the management of a software project.

Brooks (*The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. McConnell (*Software Project Survival Guide,* Microsoft Press, 1997) presents excellent pragmatic guidance for those who must manage software projects. Purba and Shah (*How to*

Manage a Successful Software Project, 2nd ed., Wiley, 2000) present a number of case studies that indicate why some projects succeed and others fail. Books by Kerzner (*Project Management: A Systems Approach to Planning Scheduling and Controlling,* 10th ed., Wiley, 2009) and Bennatan (*On Time Within Budget,* 3rd ed., Wiley, 2000) presents useful tips and guidelines for software project managers. Weigers (*Practical Project Initiation,* Microsoft Press, 2007) provides practical guidelines for getting a software project off the ground successfully.

It can be argued that the most important aspect of software project management is people management. Cockburn (*Agile Software Development,* Addison-Wesley, 2002) presents one of the best discussions of software people written to date. DeMarco and Lister [DeM98] have written the definitive book on software people and software projects. In addition, the following books on this subject have been published in recent years and are worth examining:

Cantor, M., *Software Leadership: A Guide to Successful Software Development,* Addison-Wesley, 2001.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Chandler, H. M., *Game Production Handbook,* 2nd ed., Charles River Media, 2008.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software,* Prentice Hall, 2001.

Ebert, C., *Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing,* Wiley-IEEE Computer Society, 2011.

Fairley, R. E., *Managing and Leading Software Projects,* Wiley-IEEE Computer Society, 2009.

Garton, C., and Wegryn, K., *Managing Without Walls,* McPress, 2006.

Humphrey, W. S., and Over, J. W., *Leadership, Teamwork, and Trust: Building a Competitive Software Capability,* Addison-Wesley, 2011.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W. S., *TSP-Coaching Development Teams,* Addison-Wesley, 2006.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Misrik, I., et al., *Collaborative Software Engineering,* Springer, 2010.

Peters, L., *Getting Results from Software Development Teams,* Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team,* Addison-Wesley, 2001.

Even though they do not relate specifically to the software world and sometimes suffer from oversimplification and broad generalization, best-selling "management" books by Kanter (*Confidence,* Three Rivers Press, 2006), Covy (*The 8th Habit,* Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done,* Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999), and Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy. Older titles such as *Who Moved My Cheese?*, *The One-Minute Manager,* and *In Search of Excellence* continue to provide valuable insights that can help you to manage people and projects more effectively.

A wide variety of information sources on the software project management are available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman.**

CHAPTER

32 Process and Project Metrics

Key Concepts

defect removal efficiency (DRE) . . 718
function point (FP) . 710
measurement 708
metrics
arguments for . 720
baseline. 720
establishing a program. 722
function-oriented. 710
LOC-based metrics 712
object-oriented. 713
private and public. 706
process 704
productivity . . . 712
project. 707
size-oriented . . 709

Measurement enables us to gain insight into the process and the project by providing a mechanism for objective evaluation. Lord Kelvin once said:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has taken Lord Kelvin's words to heart. But not without frustration and more than a little controversy!

Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds (Chapter 30).

Quick Look

What is it? Software process and project metrics are quantitative measures that enable you to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Who does it? Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers.

Why is it important? If you don't measure, judgment can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

What are the steps? Begin by defining a limited set of process, project, and product measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analyzed and compared to past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

What is the work product? A set of software metrics that provides insight into the process and understanding of the project.

How do I ensure that I've done it right? By applying a consistent, yet simple measurement scheme that is never to be used to assess, reward, or punish individual performance.

software quality 716
use case-oriented . . 714
WebApps 714

Within the context of the software process and the projects that are conducted using the process, a software team is concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us plan and estimate more accurately?

In their guidebook on software measurement, Park, Goethert, and Florac [Par96b] note the reasons that we measure: (1) to *characterize* in an effort to gain an understanding "of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments"; (2) to *evaluate* "to determine status with respect to plans"; (3) to *predict* by "gaining understandings of relationships among processes and products and building models of these relationships"; and (4) to *improve* by "identify[ing] roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance."

Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

32.1 Metrics in the Process and Project Domains

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement (Chapter 37). *Project metrics* enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

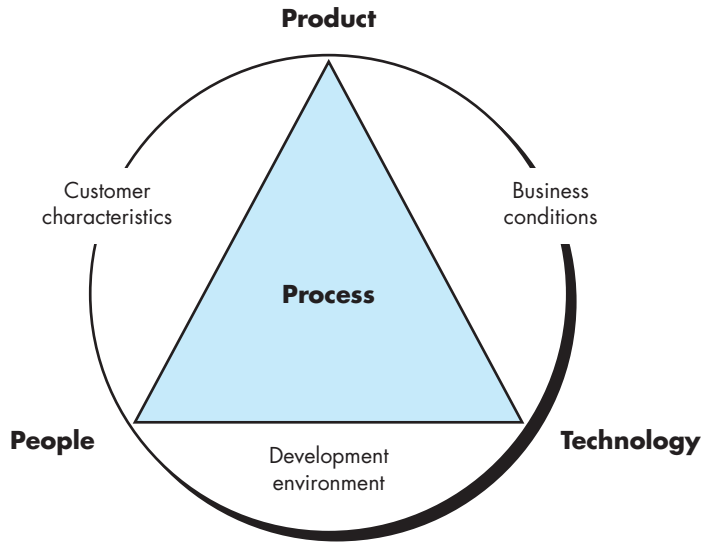
Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domains.

32.1.1 Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement (Chapter 37). But before we discuss software metrics and their impact on

FIGURE 32.1

Determinants for software quality and organizational effectiveness. *Source:* Adapted from [Pau94].



software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance" [Pau94].

KEY POINT

The skill and motivation of the software people doing the work are the most important factors that influence software quality.

Referring to Figure 32.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people have been shown [Boe81] to be the most influential factors in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

Quote:

"Software metrics let you know when to laugh and when to cry."

Tom Gilb

You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time used, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities described in Chapter 2.

? What is the difference between private and public uses for software metrics?

Grady [Gra92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of *private metrics* include defect rates (by individual), defect rates (by component), and errors found during development.

The "private process data" philosophy conforms well with the Personal Software Process approach (Chapter 4) proposed by Humphrey [Hum05]. Humphrey recognized that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as you work to improve your software engineering approach.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during technical reviews, and lines of code or function points per component or function.[1] The team reviews these data to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project-level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefits as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [Gra92] suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

? What guidelines should be applied when we collect software metrics?

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.

1 Lines of code and function point metrics are discussed in Sections 32.2.1 and 32.2.2.

- Don't obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects[2] encountered as an application, system, or product is developed and used.

32.1.2 Project Metrics

Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 30) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

? How should we use metrics during the project itself?

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

2 In this book, an *error* is defined as some flaw in a software engineering work product that is uncovered *before* the software is delivered to the end user. A *defect* is a flaw that is uncovered *after* delivery to the end user. It should be noted that others do not make this distinction.

SafeHome

Establishing a Metrics Approach

The scene: Doug Miller's office as the *SafeHome* software project is about to begin.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: Before we start work on this project, I'd like you guys to define and collect a set of simple metrics. To start, you'll have to define your goals.

Vinod (frowning): We've never done that before, and . . .

Jamie (interrupting): And based on the time line management has been talking about, we'll never have the time. What good are metrics anyway?

Doug (raising his hand to stop the onslaught): Slow down and take a breath, guys. The fact that we've never done it before is all the more reason to start now, and the metrics work I'm talking about shouldn't take much time at all . . . in fact, it just might save us time.

Vinod: How?

Doug: Look, we're going to be doing a lot more in-house software engineering as our products get more intelligent, become Web enabled, mobile, all that . . . and we need to understand the process we use to build software . . . and improve it so we can build software better. The only way to do that is to measure.

Jamie: But we're under time pressure, Doug. I'm not in favor of more paper pushing . . . we need the time to do our work, not collect data.

Doug (calmly): Jamie, an engineer's work involves collecting data, evaluating it, and using the results to improve the product and the process. Am I wrong?

Jamie: No, but . . .

Doug: What if we hold the number of measures we collect to no more than five or six and focus on quality?

Vinod: No one can argue against high quality . . .

Jamie: True . . . but, I don't know. I still think this isn't necessary.

Doug: I'm going to ask you to humor me on this one. How much do you guys know about software metrics?

Jamie (looking at Vinod): Not much.

Doug: Here are some Web refs . . . spend a few hours getting up to speed.

Jamie (smiling): I thought you said this wouldn't take any time.

Doug: Time you spend learning is never wasted . . . go do it and then we'll establish some goals, ask a few questions, and define the metrics we need to collect.

32.2 Software Measurement

Quote:

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein

In Chapter 30 we noted that measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" that are discussed in Chapter 19.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as

specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have partitioned the software metrics domain into process, project, and product metrics and noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

ADVICE

Because many factors affect software work, don't use metrics to compare individuals or teams.

To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

32.2.1 Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 32.2, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 32.2) for project *alpha*: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the

FIGURE 32.2

Size-oriented metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project *alpha* indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, you can choose lines of code as a normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per KLOC
- Pages of documentation per KLOC

In addition, other interesting metrics can be computed:

- Errors per person-month
- KLOC per person-month
- $ per page of documentation

KEY POINT

Size-oriented metrics are widely used, but debate about their validity and applicability continues.

Size-oriented metrics are not universally accepted as the best way to measure the software process. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter programs; that they cannot easily accommodate nonprocedural languages; and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

32.2.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the *function point* (FP). Computation of the function point is based on characteristics of the software's information domain and complexity. The mechanics of FP computation have been discussed in Chapter 30.[3]

3 See Section 30.2.1 for a detailed discussion of FP computation.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language–independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

32.2.3 Reconciling LOC and FP Metrics

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The following table[4] [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

Programming Language	LOC per Function Point			
	Avg.	Median	Low	High
Ada	154	—	104	205
ASP	56	50	32	106
Assembler	337	315	91	694
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	704
COBOL	80	78	8	400
ColdFusion	68	56	52	105
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Focus	43	42	32	56
FORTRAN	90	118	35	—
FoxPro	32	35	25	35
HTML	43	42	35	53
Informix	42	31	24	57
J2EE	57	50	50	67
Java	55	53	9	214
JavaScript	54	55	45	63
JSP	59	—	—	—
Lotus Notes	23	21	15	46

4 The data presented in the table is an abbreviated version of data developed by Quantitative Software Management (**www.qsm.com**) and is used with their permission, copyright 2002.

Programming Language	LOC per Function Point			
	Avg.	Median	Low	High
Mantis	71	27	22	250
Natural	51	53	34	60
.NET	60	60	60	60
Oracle	42	29	12	217
OracleDev2K	35	30	23	100
PeopleSoft	37	32	34	40
Perl	57	57	45	60
PL/1	58	57	27	92
Powerbuilder	28	22	8	105
RPG II/III	61	49	24	155
SAS	50	35	33	49
Smalltalk	26	19	10	55
SQL	31	37	13	80
VBScript	38	37	29	50
Visual Basic	50	52	14	276

A review of these data indicates that one LOC of C++ provides approximately 2.4 times the "functionality" (on average) as one LOC of C. Furthermore, one LOC of a Smalltalk provides at least four times the functionality of an LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to "backfire" [Jon98] existing software to estimate the number of function points, once the total number of programming language statements are known.

LOC and FP measures are often used to derive *productivity metrics*. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answer to these questions is an emphatic no! The reason for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Function points and *LOC-based metrics* have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 33), an historical baseline of information must be established.

Within the context of process and project metrics, you should be concerned primarily with productivity and quality—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For process improvement and project planning purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that

was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

32.2.4 Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process. Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:

ADVICE

It is not uncommon for multiple-scenario scripts to mention the same functionality or data objects. Therefore, be careful when using script counts. Many scripts can sometimes be reduced to a single class or set of code.

Number of scenario scripts. A scenario script (analogous to a use case) is a detailed sequence of steps that describes the interaction between the user and the application. Each script is organized into triplets of the form

{**initiator,** *action,* **participant**}

where **initiator** is the object that requests some service (that initiates a message), *action* is the result of the request, and **participant** is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

ADVICE

Classes can vary in size and complexity. Therefore, it's worth considering classifying class counts by size and complexity.

Number of key classes. *Key classes* are the "highly independent components" [Lor94] that are defined early in object-oriented analysis (Chapter 10).[5] Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Number of support classes. *Support classes* are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (UI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Average number of support classes per key class. In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified.

5 We referred to key classes as *analysis classes* in Chapter 10.

Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

Number of subsystems. A *subsystem* is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

To be used effectively in an object-oriented software engineering environment, metrics similar to those noted above must be collected along with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the object-oriented measures and project measures will provide metrics that can aid in project estimation.

32.2.5 Use Case-Oriented Metrics

Use cases[6] are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP. Like FP, the use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated. Use cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use cases can be created at vastly different levels of abstraction, there is no standard "size" for a use case. Without a standard "measure" of what a use case is, its application as a normalization measure (e.g., effort expended per use case) is suspect.

Researchers have suggested *use-case points* (UCPs) as a mechanism for estimating project effort and other characteristics. The UCP is a function of the number of actors and transactions implied by the use-case models and is analogous to the FP in some ways. If you have further interest, see [Coh05], [Cle06], or [Col09].

32.2.6 WebApp Project Metrics

The objective of all WebApp projects is to deliver a combination of content and functionality to the end user. Measures and metrics used for traditional software engineering projects are difficult to translate directly to WebApps. Yet, it is possible to develop a database that allows assess to internal productivity and quality

6 Use cases are introduced in Chapters 8 and 9.

measures derived over a number of projects. Among the measures that can be collected are the following:

Number of static Web pages. These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

Number of dynamic Web pages. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

Number of internal page links. This measure provides an indication of the degree of architectural coupling within the WebApp. As the number of page links increases, the effort expended on navigational design and construction also increases.

Number of persistent data objects. As the number of persistent data objects (e.g., a database or data file) grows, the complexity of the WebApp also grows and the effort to implement it increases proportionally.

Number of external systems interfaced. As the requirement for interfacing grows, system complexity and development effort also increase.

Number of static content objects. These objects represent low relative complexity and generally require less effort to construct than dynamic pages.

Number of dynamic content objects. These objects represent higher relative complexity and require more effort to construct than static pages.

Number of executable functions. As the number of executable functions (e.g., a script or applet) increases, modeling and construction effort also increase.

Each of the preceding measures can be determined at a relatively early stage. For example, you can define a metric that reflects the degree of end-user customization that is required for the WebApp and correlate it to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted. To accomplish this, you define

N_{sp} = number of static Web pages

N_{dp} = number of dynamic Web pages

Then,

$$\text{Customization index, } C = \frac{N_{dp}}{(N_{dp} + N_{sp})}$$

The value of C ranges from 0 to 1. As C grows larger, the level of WebApp customization becomes a significant technical issue.

Similar WebApp metrics can be computed and correlated with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the WebApp measures and project measures will provide indicators that can aid in project estimation.

SOFTWARE TOOLS

Project and Process Metrics

Objective: To assist in the definition, collection, evaluation, and reporting of software measures and metrics.

Mechanics: Each tool varies in its application, but all provide mechanisms for collecting and evaluating data that lead to the computation of software metrics.

Representative Tools:[7]

Function Point WORKBENCH, developed by Charismatek (**www.charismatek.com.au**), offers a wide array of FP-oriented metrics.

DataDrill, developed by Distributive Software (**www.distributive.com**), supports automating data collection, analysis, chart formatting, report generation, and other measurement tasks.

PSM Insight, developed by Practical Software and Systems Measurement (**www.psmsc.com**), assists in the creation and subsequent analysis of a project measurement database.

SLIM tool set, developed by QSM (**www.qsm.com**), provides a comprehensive set of metrics and estimation tools.

SPR tool set, developed by Software Productivity Research (**www.spr.com**), offers a comprehensive collection of FP-oriented tools.

TychoMetrics, developed by Predicate Logic (**www.predicate.com**), is a tool suite for management metrics collection and reporting.

32.3 METRICS FOR SOFTWARE QUALITY

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. You can use measurement to assess the quality of the requirements and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time assessment, you apply product metrics (Chapter 30) to evaluate the quality of software engineering work products in objective, rather than subjective ways.

ADVICE

Software is a complex entity. Therefore, errors are to be expected as work products are developed. Process metrics are intended to improve the software process so that errors are uncovered in the most effective manner.

A project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are combined to provide project-level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

Metrics such as work product errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed in Section 32.3.3.

32.3.1 Measuring Quality

WebRef

An excellent source of information on software quality and related topics (including metrics) can be found at **http://searchsoftwarequality.techtarget.com/resources.**

Although there are many measures of software quality,[8] correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [Gil88] suggests definitions and measures for each.

Correctness. Correctness is the degree to which the software performs its required function. Defects (lack of correctness) are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.

Maintainability. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

Integrity. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as:

$$\text{Integrity} = \Sigma\,[1 - (\text{threat} \times (1 - \text{security}))]$$

For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high). If, on the other hand, the threat probability is

8 A detailed discussion of the factors that influence software quality and the metrics that can be used to assess software quality has been presented in Chapter 30.

0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

Usability. Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics presented in Chapter 15.

These four factors are only a sampling of those that have been proposed as measures for software quality. Chapter 30 considers this topic in additional detail.

32.3.2 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is *defect removal efficiency* (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = \frac{E}{E + D}$$

where E is the number of errors found before delivery of the software to the end user and D is the number of defects found after delivery.

ADVICE

If DRE is low as you move through analysis and design, spend some time improving the way you conduct formal technical reviews.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found). When used in this context, we redefine DRE as

$$\text{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$

where E_i is the number of errors found during software engineering action i and E_i+_1 is the number of errors found during software engineering action $i + 1$ that are traceable to errors that were not discovered in software engineering action i.

A quality objective for a software team (or an individual software engineer) is to achieve DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity or action.

SafeHome

Establishing a Metrics Approach

The scene: Doug Miller's office two days after initial meeting on software metrics.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman and Jamie Lazar, members of the product software engineering team.

The conversation:

Doug: You both had a chance to learn a little about process and project metrics?

Vinod and Jamie: [Both nod.]

Doug: It's always a good idea to establish goals when you adopt any metrics. What are yours?

Vinod: Our metrics should focus on quality. In fact, our overall goal is to keep the number of errors we pass on from one software engineering activity to the next to an absolute minimum.

Doug: And be very sure you keep the number of defects released with the product to as close to zero as possible.

Vinod (nodding): Of course.

Jamie: I like DRE as a metric, and I think we can use it for the entire project, but also as we move from one framework activity to the next. It'll encourage us to find errors at each step.

Vinod: I'd also like to collect the number of hours we spend on reviews.

Jamie: And the overall effort we spend on each software engineering task.

Doug: You can compute a review-to-development ratio . . . might be interesting.

Jamie: I'd like to track some use-case data as well. Like the amount of effort required to develop a use case, the amount of effort required to build software to implement a use case, and . . .

Doug (smiling): I thought we were going to keep this simple.

Vinod: We should, but once you get into this metrics stuff, there's a lot of interesting things to look at.

Doug: I agree, but let's walk before we run and stick to our goal. Limit data to be collected to five or six items, and we're ready to go.

32.4 Integrating Metrics within the Software Process

The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted previously in this chapter, the problem is cultural. Attempting to collect measures where none have been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner.

In this section, we consider some arguments for software metrics and present an approach for instituting a metrics collection program within a software engineering organization. But before we begin, some words of wisdom (now almost three decades old) are offered by Grady and Caswell [Gra87]:

> Some of the things we describe here will sound quite easy. Realistically, though, establishing a successful company-wide software metrics program is hard work. When we say that you must wait at least three years before broad organizational trends are available, you get some idea of the scope of such an effort.

The caveat suggested by the authors is well worth heeding, but the benefits of measurement are so compelling that the hard work is worth it.

32.4.1 Arguments for Software Metrics

Quote:

"We manage things by the numbers in many aspects of our lives . . . These numbers give us insight and help steer our actions."

Michael Mah and Larry Putnam

Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If you do not measure, there no real way of determining whether you are improving. And if you are not improving, you are lost.

By requesting and evaluating productivity and quality measures, a software team (and its management) can establish meaningful goals for improvement of the software process. Early in this book, we noted that software is a strategic business issue for many companies. If the process through which it is developed can be improved, a direct impact on the bottom line can result. But to establish goals for improvement, the current status of software development must be understood. Hence, measurement is used to establish a process baseline from which improvements can be assessed.

The day-to-day rigors of software project work leave little time for strategic thinking. Software project managers are concerned with more mundane (but equally important) issues: developing meaningful project estimates, producing higher quality systems, getting product out the door on time. By using measurement to establish a project baseline, each of these issues becomes more manageable. We have already noted that the baseline serves as a basis for estimation. Additionally, the collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.[9]

32.4.2 Establishing a Baseline

By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 32.2 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

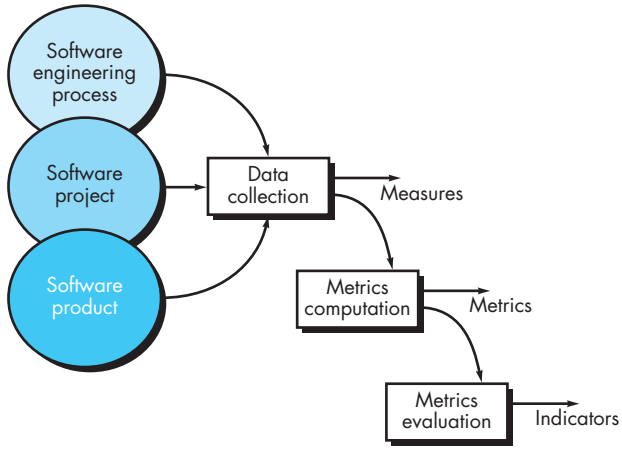
? What is a metrics baseline and what benefit does it provide to a software engineer?

To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes: (1) data must be reasonably accurate—"guestimates" about past projects are to be avoided, (2) data should be collected for as many projects as possible, (3) measures must be consistent (for example, a line of code must be interpreted consistently across all projects for which data are collected), (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

9 These ideas have been formalized into an approach called *statistical software quality assurance*.

FIGURE 32.3

Software metrics collection process



32.4.3 Metrics Collection, Computation, and Evaluation

ADVICE

If you're just starting to collect metrics data, remember to keep it simple. If you bury yourself with data, your metrics effort will fail.

The process for establishing a metrics baseline is illustrated in Figure 32.3. Ideally, data needed to establish a baseline have been collected in an ongoing manner. Sadly, this is rarely the case. Therefore, data collection requires an historical investigation of past projects to reconstruct required data. Once measures have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of measures collected, metrics can span a broad range of application-oriented metrics (e.g., LOC, FP, object-oriented, WebApp) as well as other quality- and project-oriented metrics. Finally, metrics must be evaluated and applied during estimation, technical work, project control, and process improvement. Metrics evaluation focuses on the underlying reasons for the results obtained and produces a set of indicators that guide the project or process.

32.5 METRICS FOR SMALL ORGANIZATIONS

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it is reasonable to suggest that software organizations[10] of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce.

A small organization can begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires

10 This discussion is equally relevant to software teams that have adopted an agile software development process (Chapter 5).

improvement. For example, "reduce the time to evaluate and implement change requests." A small organization might select the following set of easily collected measures:

? How should we derive a set of "simple" software metrics?

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, t_{queue}.
- Effort (person-hours) to perform the evaluation, W_{eval}.
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, t_{eval}.
- Effort (person-hours) required to make the change, W_{change}.
- Time required (hours or days) to make the change, t_{change}.
- Errors uncovered during work to make change, E_{change}.
- Defects uncovered after change is released to the customer base, D_{change}.

Once these measures have been collected for a number of change requests, it is possible to compute the total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data, E_{change} and D_{change}. The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce t_{queue}, W_{eval}, t_{eval}, W_{change}, and/or E_{change}. In addition, the defect removal efficiency can be computed as

$$\text{DRE} = \frac{E_{change}}{E_{change} + D_{change}}$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

32.6 Establishing a Software Metrics Program

The Software Engineering Institute has developed a comprehensive guidebook [Par96b] for establishing a "goal-driven" software metrics program. The guidebook suggests the following steps: (1) identify your business goals; (2) identify what you want to know or learn; (3) identify your subgoals; (4) identify the entities and attributes related to your subgoals; (5) formalize your measurement goals; (6) identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals; (7) identify the data elements that you will collect to construct the indicators; (8) identify the measures to be used, and make these definitions operational; (9) identify the actions that you will take to implement the measures, and (10) prepare a plan for implementing the

measures. A detailed discussion of these steps is best left to the SEI's guidebook. However, a brief overview of key points is worthwhile.

KEY POINT
The software metrics you choose should be driven by the business and technical goals you wish to accomplish.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider the *SafeHome* product. Working as a team, software engineering and business managers develop a list of prioritized business goals:

1. Improve our customers' satisfaction with our products.
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

The software organization examines each business goal and asks: "What activities do we manage, execute, or support and what do we want to improve within these activities?" To answer these questions the SEI recommends the creation of an "entity-question list" in which all things (entities) within the software process that are managed or influenced by the software organization are noted. Examples of entities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process.

Consider the fourth goal: "Make support for our products easier." The following list of questions might be derived for this goal [Par96b]:

- Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?
- How large is the change request backlog?
- Is our response time for fixing bugs acceptable based on customer need?
- Is our change control process (Chapter 29) followed?
- Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following subgoal: *Improve the performance of the change management process*. The software process entities and attributes that are relevant to the subgoal are identified, and the measurement goals associated with them are delineated.

The SEI [Par96b] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, you refine measurement goals into questions that are further refined into entities and attributes that are then refined into metrics.

32.7 Summary

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of software projects; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project work flow and technical approach in a real-time manner.

Both size- and function-oriented metrics are used throughout the industry. Size-oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. The function point is derived from measures of the information domain and a subjective assessment of problem complexity. In addition, object-oriented metrics and Web application metrics can be used.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business. By creating a metrics baseline—a database containing process and product measurements—software engineers and their managers can gain better insight into the work that they do and the product that they produce.

Problems and Points to Ponder

32.1. Describe the difference between process and project metrics in your own words.

32.2. Why should some software metrics be kept "private"? Provide examples of three metrics that should be private. Provide examples of three metrics that should be public.

32.3. What is an indirect measure, and why are such measures common in software metrics work?

32.4. Grady suggests an etiquette for software metrics. Can you add three more rules to those noted in Section 32.1.1?

32.5. Team A found 342 errors during the software engineering process prior to release. Team B found 184 errors. What additional measures would have to be made for projects A and B to determine which of the teams eliminated errors more efficiently? What metrics would you propose to help in making the determination? What historical data might be useful?

32.6. Present an argument against lines of code as a measure for software productivity. Will your case hold up when dozens or hundreds of projects are considered?

32.7. Compute the function point value for a project with the following information domain characteristics:

Number of user inputs: 32
Number of user outputs: 60
Number of user inquiries: 24
Number of files: 8
Number of external interfaces: 2

Assume that all complexity adjustment values are average. Use the algorithm noted in Chapter 30.

32.8. Using the table presented in Section 32.2.3, make an argument against the use of assembler language based on the functionality delivered per statement of code. Again referring to the table, discuss why C++ would present a better alternative than C.

32.9. The software used to control a photocopier requires 32,000 lines of C and 4,200 lines of Smalltalk. Estimate the number of function points for the software inside the copier.

32.10. A Web engineering team has built an e-commerce WebApp that contains 145 individual pages. Of these pages, 65 are dynamic; that is, they are internally generated based on end-user input. What is the customization index for this application?

32.11. A WebApp and its support environment have not been fully fortified against attack. Web engineers estimate that the likelihood of repelling an attack is only 30 percent. The system does not contain sensitive or controversial information, so the threat probability is 25 percent. What is the integrity of the WebApp?

32.12. At the conclusion of a project, it has been determined that 30 errors were found during the modeling phase and 12 errors were found during the construction phase that were traceable to errors that were not discovered in the modeling phase. What is the DRE for these two phases?

32.13. A software team delivers a software increment to end users. The users uncover eight defects during the first month of use. Prior to delivery, the software team found 242 errors during formal technical reviews and all testing tasks. What is the overall DRE for the project after one month's usage?

Further Readings and Information Sources

Software process improvement (SPI) has received a significant amount of attention over the past two decades. Since measurement and software metrics are key to successfully improving the software process, many books on SPI also discuss metrics. Books by Arban (*Software Metrics and Software Methodology,* Wiley-IEEE Computer Society, 2010), Rico (*ROI of Software Process Improvement,* J. Ross Publishing, 2004) provides an in-depth discussion of SPI and the metrics that can help an organization achieve it. Ebert and his colleagues (*Best Practices in Software Measurement,* Springer, 2004) address the use of measurement within

the context of ISO and CMMI standards. Kan (*Metrics and Models in Software Quality Engineering,* 2nd ed., Addison-Wesley, 2002) presents a collection of relevant metrics.

Ebert and Dumke (*Software Measurement,* Springer, 2007) provide a useful treatment of measurement and metrics as they should be applied for IT projects. McGarry and his colleagues (*Practical Software Measurement,* Addison-Wesley, 2001) present in-depth advice for assessing the software process. A worthwhile collection of papers has been edited by Haug and his colleagues (*Software Process Improvement: Metrics, Measurement, and Process Modeling,* Springer-Verlag, 2001). Florac and Carlton (*Measuring the Software Process,* Addison-Wesley, 1999) and Fenton and Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Revised, Brooks/Cole Publishers, 1998) discuss how software metrics can be used to provide the indicators necessary to improve the software process.

Wohlin and his colleagues (*Experimentation in Software Engineering,* Springer, 2012) discusses the manner in which measures used in analyzing software process. Jones (*Applied Software Measurement: Global Analysis of Productivity and Quality,* McGraw-Hill, 2008), Laird and Brennan (*Software Measurement and Estimation,* Wiley-IEEE Computer Society Press, 2006), and Goodman (*Software Metrics: Best Practices for Successful IT Management*, Rothstein Associates, 2004) discuss the use of software metrics for project management and estimation. Putnam and Myers (*Five Core Metrics,* Dorset House, 2003) draw on a database of more than the 6,000 software projects to demonstrate how five core metrics—time, effort, size, reliability, and process productivity—can be used to control software projects. Maxwell (*Applied Statistics for Software Managers,* Prentice Hall, 2003) presents techniques for analyzing software project data. Munson (*Software Engineering Measurement,* Auerbach, 2003) discusses a broad array of software engineering measurement issues. Jones (*Software Assessments, Benchmarks and Best Practices,* Addison-Wesley, 2000) describes both quantitative measurement and qualitative factors that help an organization assess its software process and practices.

Function point measurement has become a widely used technique in many areas of software engineering work. The International Function Point Users Group published a collection of papers on the use of function point metrics (*The IFPUC Guide to IT and Software Measurement,* Auerbach, 2012). Parthasarathy (*Practical Software Estimation: Function Point Methods for Insourced and Outsourced Projects,* Addison-Wesley, 2007) provide a comprehensive guide. Garmus and Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects,* Addison-Wesley, 2000) discuss process metrics with an emphasis on function point analysis.

Relatively little has been published on metrics for Web engineering work, Clifton (*Advanced Web Metrics with Google Analytics,* 3rd ed., Sybex, 2012), However, Kaushik (*Web Analytics 2.0: Accountability and Science of Customer Centricity*, Sybex, 2009) and *(Web Analytics: An Hour a Day,* Sybex, 2007), Stern (*Web Metrics: Proven Methods for Measuring Web Site Success,* Wiley, 2002), Inan and Kean (*Measuring the Success of Your Website,* Longman, 2002), and Nobles and Grady (*Web Site Analysis and Reporting,* Premier Press, 2001) address Web metrics from a business and marketing perspective.

The latest research in the metrics area is summarized by the IEEE (*Symposium on Software Metrics,* published yearly). A wide variety of information sources on the process and project metrics is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

33 Estimation for Software Projects

Key Concepts

estimation
agile development . . 746
decomposition techniques 734
empirical models. 743
FP-based 738
make-buy 748
object-oriented projects 746
outsourcing . . . 750
problem-based . 735
process-based . 739
reconciliation . . 742

Software project management begins with a set of activities that are collectively called *project planning*. Before the project can begin, the software team should estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish. Once these activities are accomplished, the software team should establish a project schedule that defines software engineering tasks and milestones, identifies who is responsible for conducting each task, and specifies the intertask dependencies that may have a strong bearing on progress.

In an excellent guide to "software project survival," Steve McConnell [McC98] presents a real-world view of project planning:

> Many technical workers would rather do technical work than spend time planning. Many technical managers do not have sufficient training in technical management to feel confident that their planning will improve a project's outcome. Since neither party wants to do planning, it often doesn't get done.

Quick Look

What is it? A real need for software has been established; stakeholders are onboard, software engineers are ready to start, and the project is about to begin. But how do you proceed? Software project planning encompasses five major activities—estimation, scheduling, risk analysis, quality management planning, and change management planning. In the context of this chapter, we consider only estimation—your attempt to determine how much money, effort, resources, and time it will take to build a specific software-based system or product.

Who does it? Software project managers—using information solicited from project stakeholders and software metrics data collected from past projects.

Why is it important? Would you build a house without knowing how much you were about to spend, the tasks that you need to perform, and the time line for the work to be conducted? Of course not, and since most computer-based systems and products cost considerably more to build than a large house, it would seem reasonable to develop an estimate before you start creating the software.

What are the steps? Estimation begins with a description of the scope of the product. The problem is then decomposed into a set of smaller problems, and each of these is estimated using historical data and experience as guides. Problem complexity and risk are considered before a final estimate is made.

What is the work product? A simple table delineating the tasks to be performed, the functions to be implemented, and the cost, effort, and time involved for each is generated.

How do I ensure that I've done it right? That's hard, because you won't really know until the project has been completed. However, if you have experience and follow a systematic approach, generate estimates using solid historical data, create estimation data points using at least two different methods, establish a realistic schedule, and continually adapt it as the project moves forward, you can feel confident that you've given it your best shot.

use cases 740
WebApps 747
project planning . . 729
resources 731
software equation 744
software scope . . . 730
software sizing . . . 734
use case points (UCPs) 742

> But failure to plan is one of the most critical mistakes a project can make . . . effective planning is needed to resolve problems upstream [early in the project] at low cost, rather than downstream [late in the project] at high cost. The average project spends *80 percent* of its time on rework—fixing mistakes that were made earlier in the project.

McConnell argues that every project can find the time to plan (and to adapt the plan throughout the project) simply by taking a small percentage of the time that would have been spent on rework that occurs because planning was not conducted.

33.1 Observations on Estimation

Planning requires you to make an initial commitment, even though it's likely that this "commitment" will be proven wrong. Whenever estimates are made, you look into the future and accept some degree of uncertainty as a matter of course. To quote Frederick Brooks [Bro95]:

> . . . our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption that is quite untrue, i.e., that all will go well. . . . because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product.

Quote:

"Good estimating approaches and solid historical data offer the best hope that reality will win out over impossible demands."

Caper Jones

Although estimating is as much art as it is science, this important action need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates. Past experience (of all people involved) can aid immeasurably as estimates are developed and reviewed. Because estimation lays a foundation for all other project planning actions, and project planning provides the road map for successful software engineering, we would be ill advised to embark without it.

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk,[1] and this risk leads to uncertainty.

Project complexity has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past effort. The first-time developer of a sophisticated e-commerce application might consider it to be exceedingly complex. However, a Web engineering team developing its tenth e-commerce WebApp would consider such work run of the mill. A number of quantitative software complexity measures have been proposed [Zus97]. Such measures are applied at the design or code level and

1 Systematic techniques for risk analysis are presented in Chapter 35.

are therefore difficult to use during software planning (before a design and code exist). However, other, more subjective assessments of complexity (e.g., function point complexity adjustment factors described in Chapter 30) can be established early in the planning process.

KEY POINT

Project complexity, project size, and the degree of structural uncertainty all affect the reliability of estimates.

Project size is another important factor that can affect the accuracy and efficacy of estimates. As size increases, the interdependency among various elements of the software grows rapidly.[2] Problem decomposition, an important approach to estimating, becomes more difficult because the refinement of problem elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail.

The *degree of structural uncertainty* also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information that must be processed.

Quote:

"It is the mark of an instructed mind to rest satisfied with the degree of precision that the nature of the subject admits, and not to seek exactness when only an approximation of the truth is possible."

Aristotle

The availability of historical information has a strong influence on estimation risk. By looking back, you can emulate things that worked and improve areas where problems arose. When comprehensive software metrics (Chapter 32) are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high. As a planner, you and the customer should recognize that variability in software requirements means instability in cost and schedule.

However, you should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible—although not always politically acceptable—to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

33.2 The Project Planning Process

ADVICE

The more you know, the better you estimate. Therefore, update your estimates as the project progresses.

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be adapted and updated

2 Size often increases due to "scope creep" that occurs when problem requirements change. Increases in project size can have a geometric impact on project cost and schedule (Michael Mah, personal communication).

as the project proceeds. In the following sections, each of the activities associated with software project planning is discussed.

TASK SET

Task Set for Project Planning

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks (Chapter 35).
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.
5. Estimate cost and effort.
   a. Decompose the problem.
   b. Develop two or more estimates using size, function points, process tasks, or use cases.
   c. Reconcile the estimates.
6. Develop a project schedule (Chapter 34).
   a. Establish a meaningful task set.
   b. Define a task network.
   c. Use scheduling tools to develop a time-line chart.
   d. Define schedule tracking mechanisms.

33.3 SOFTWARE SCOPE AND FEASIBILITY

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that *bound* the system. Scope is defined using one of two techniques:

ADVICE

Project feasibility is important, but a consideration of business need is even more important. It does no good to build a high-tech system or product that no one wants.

1. A narrative description of software scope is developed after communication with all stakeholders.
2. A set of use cases[3] is developed by end users.

Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?" All too often, software engineers rush past these questions (or are pushed past them by impatient managers or other stakeholders), only to become mired in a project that is doomed from the onset.

3 Use cases have been discussed in detail throughout Part 2 of this book. A use case is a scenario-based description of the user's interaction with the software from the user's point of view.

Putnam and Myers [Put97a] suggest that scoping is not enough. Once scope is understood, you must work to determine if it can be accomplished within the dimensions of available technology, dollars, time, and resources. This is a crucial, although often overlooked, part of the estimation process.

33.4 Resources

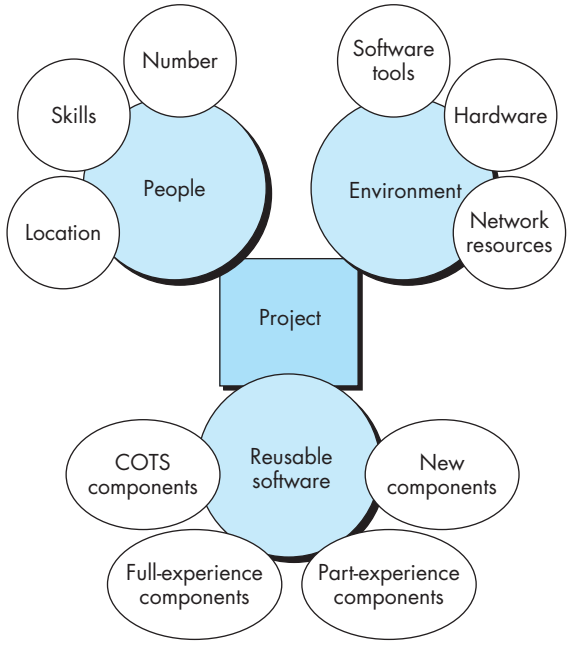
The second planning task is estimation of the resources required to accomplish the software development effort. Figure 33.1 depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a *time window*. Availability of the resource for a specified window must be established at the earliest practical time.

33.4.1 Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specified. For relatively small projects (a few person-months),

Figure 33.1

Project resources



a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified.

The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

33.4.2 Reusable Software Resources

Component-based software engineering (CBSE)[4] emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests four software resource categories that should be considered as planning proceeds: *off-the-shelf components* (existing software that can be acquired from a third party or from a past project), *full-experience components* (existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project), *partial-experience components* (existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification), and *new components* (components built by the software team specifically for the needs of the current project).

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

33.4.3 Environmental Resources

The environment that supports a software project, often called the *software engineering environment* (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.[5] Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware

4 CBSE was considered in Chapter 14.

5 Other hardware—the *target environment*—is the computer on which the software will execute when it has been released to the end user.

elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.

33.5 Software Project Estimation

Quote:

"In an age of outsourcing and increased competition, the ability to estimate more accurately . . . has emerged as a critical success factor for many IT groups."

Rob Thomsett

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

Quote:

"It is very difficult to make a vigorous, plausible and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers."

Fred Brooks

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form

$$d = f(v_i)$$

where d is one of a number of estimated values (e.g., effort, cost, project duration) and v_i are selected independent parameters (e.g., estimated LOC or FP).

Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 32, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.

33.6 Decomposition Techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

In Chapter 31, the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

33.6.1 Software Sizing

KEY POINT

The "size" of software to be built can be estimated using a direct measure, LOC, or an indirect measure, FP.

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which you have properly estimated the size of the product to be built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

Because a project estimate is only as good as the estimate of the size of the work to be accomplished, *software sizing* represents your first major challenge as a planner. In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP). Size can be estimated by considering the type of project and its application domain, the functionality delivered (i.e., the number of function points), the number of components to be delivered, the degree to which a set of existing components must be modified for the new system.

Putnam and Myers [Put92] suggest that the results of each of these sizing approaches be combined statistically to create a *three-point* or *expected-value* estimate. This is accomplished by developing optimistic (low), most likely, and pessimistic (high) values for size and combining them using Equation (33.1), described in Section 33.6.2.

33.6.2 Problem-Based Estimation

In Chapter 32, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

? What do LOC- and FP-based estimation have in common?

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

ADVICE

When collecting productivity metrics for projects, be sure to establish a taxonomy of project types. This will enable you to compute domain-specific averages, making estimation more accurate.

Baseline productivity metrics (e.g., LOC/pm or FP/pm)[6] are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project.

It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single-baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment

6 The acronym *pm* means person-month of effort.

values discussed in Chapter 30—are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

? How do we compute the "expected value" for software size?

A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates. For example,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6} \tag{33.1}$$

gives heaviest credence to the "most likely" estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is, we can't be sure. Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

33.6.3 An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, we consider a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on a desktop workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer. A preliminary statement of software scope can be developed:

> The mechanical CAD software will accept two- and three-dimensional geometric data from a designer. The designer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of devices. The software will be designed to control and interact with peripheral devices that include a mouse, scanner, laser printer, and plotter.

ADVICE

Many modern applications reside on a network or are part of a client-server architecture. Therefore, be sure that your estimates include the effort required to develop "infrastructure" software.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin, the planner must determine what

FIGURE 33.2

Estimation table for the LOC methods

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

"characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

For our purposes, assume that further refinement has occurred and that the major software functions listed in Figure 33.2 are identified. Following the decomposition technique for LOC, an estimation table (Figure 33.2) is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC; and pessimistic, 8600 LOC. Applying Equation (33.1), the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.

ADVICE

Do not succumb to the temptation to use this result as your project estimate. You should derive another result using a different approach.

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of $8,000 per month, the cost per line of code is approximately $13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $431,000 and the estimated effort is 54 person-months.[7]

SAFEHOME

Estimating

The scene: Doug Miller's office as project planning begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: We need to develop an effort estimate for the project and then we've got to define a micro schedule for the first increment and a macro schedule for the remaining increments.

7 Estimates are rounded to the nearest $1,000 and person-month. Further precision is unnecessary and unrealistic, given the limitations of estimation accuracy.

Vinod (nodding): Okay, but we haven't defined any increments yet.

Doug: True, but that's why we need to estimate.

Jamie (frowning): You want to know how long it's going to take us?

Doug: Here's what I need. First, we need to functionally decompose the *SafeHome* software . . . at a high level . . . then we've got to estimate the number of lines of code that each function will take . . . then . . .

Jamie: Whoa! How are we supposed to do that?

Vinod: I've done it on past projects. You begin with use cases, determine the functionality required to implement each, then guesstimate the LOC count for each piece of the function. The best approach is to have everyone do it independently and then compare results.

Doug: Or you can do a functional decomposition for the entire project.

Jamie: But that'll take forever and we've got to get started.

Vinod: No . . . it can be done in a few hours . . . this morning, in fact.

Doug: I agree . . . we can't expect exactitude, just a ballpark idea of what the size of *SafeHome* will be.

Jamie: I think we should just estimate effort . . . that's all.

Doug: We'll do that too. Then use both estimates as a cross-check.

Vinod: Let's go do it . . .

33.6.4 An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table presented in Figure 33.3, you would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. An FP value is computed using the technique discussed in Chapter 30. For the purposes of this estimate, the complexity weighting factor is assumed to be average. Figure 33.3 presents the results of this estimate.

Each of the complexity weighting factors is estimated, and the value adjustment factor is computed as described in Chapter 30:

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Value adjustment factor	**1.17**

FIGURE 33.3

Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
Count total						*320*

Finally, the estimated number of FP is derived:

$$FP_{estimated} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] = 375$$

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of $8,000 per month, the cost per FP is approximately $1,230. Based on the FP estimate and the historical productivity data, the total estimated project cost is $461,000 and the estimated effort is 58 person-months.

33.6.5 Process-Based Estimation

ADVICE

If time permits, use finer granularity when specifying tasks in Figure 33.4. For example, break analysis into its major tasks and estimate each separately.

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of activities, actions, and tasks and the effort required to accomplish each is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities[8] may be represented as part of a table similar to the one presented in Figure 33.4.

Once problem functions and process activities are melded, you estimate the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 33.4. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity.

If process-based estimation is performed independently of LOC or FP estimation, you now have two or three estimates for cost and effort that may be

8 The framework activities chosen for this project differ somewhat from the generic activities discussed in Chapter 3. They are: customer communication (CC), planning, risk analysis, engineering, and construction/release.

FIGURE 33.4

Process-based estimation table

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

33.6.6 An Example of Process-Based Estimation

To illustrate the use of process-based estimation, we again consider the CAD software introduced in Section 33.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

Quote:

"It's best to understand the background of an estimate before you use it."

Barry Boehm and Richard Fairley

Referring to the completed process-based table shown in Figure 33.4, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of $8,000 per month, the total estimated project cost is $368,000 and the estimated effort is 46 person-months. If desired, labor rates could be associated with each framework activity or software engineering task and computed separately.

33.6.7 Estimation with Use Cases

As we have noted throughout Part 2 of this book, use cases provide a software team with insight into software scope and requirements. Once use cases have

? Why is it difficult to develop an estimation technique using use cases?

been developed they can be used to estimate the projected "size" of a software project. However, developing an estimation approach with use cases presents challenges [Smi99]. Use cases are described using many different formats and styles and represent an external view (the user's view) of the software. Therefore, they can be written at many different levels of abstraction. Use cases do not address the complexity of the functions and features that are described, and they can describe complex behavior (e.g., interactions) that involve many functions and features.

Even with these constraints, it is possible to compute *use case points* (UCPs) in a manner that is analogous to the computation of functions points (Chapter 30).

Cohn (Coh05) indicates that the computation of use case points must take the following characteristics into account:

- The number and complexity of the use cases in the system.
- The number and complexity of the actors on the system.
- Various nonfunctional requirements (such as portability, performance, maintainability) that are not written as use cases.
- The environment in which the project will be developed (e.g., the programming language, the software team's motivation).

To begin, each use case is assessed to determine its relative complexity. A simple use case indicates a simple user interface, a single database, and three or fewer transactions and five or fewer class implementations. An average use case indicates a more complex UI, 2 or 3 databases, and 4 to 7 transactions with 5 to 10 classes. Finally, a complex use case implies a complex UI with multiple databases, using eight or more transactions and 11 or more classes. Each use case is assessed using these criteria and the count of each type is weighted by a factor of 5, 10, and 15, respectively. A total *unadjusted use case weight* (UUCW) is the sum of all weighted counts [Nun11].

Next, each actor is assessed. Simple actors are automatons (another system, a machine or device) that communicate through an API. Average actors are automatons that communicate through a protocol or a data store, and complex actors are humans who communicate through a GUI or other human interface. Each actor is assessed using these criteria and the count of each type is weighted by a factor of 1, 2, and 3, respectively. The total *unadjusted actor weight* (UAW) is the sum of all weighted counts.

These unadjusted values are modified by considering technical complexity factors (TCFs) and environment complexity factors (ECFs). Thirteen factors contribute to an assessment of the final TCF, and eight factors contribute to the computation of the final ECF [Coh05]. Once these values have been determined, the final UCP value is computed in the following manner:

$$\text{UCP} = (\text{UUCW} + \text{UAW}) \times \text{TCF} \times \text{ECF} \tag{33.2}$$

33.6.8 An Example of Estimation Using Use Case Points

The CAD software introduced in Section 33.6.3 is composed of three subsystem groups: user interface subsystem (includes UICF), engineering subsystem group (includes the 2DGA, 3DGA, and DAM subsystems), and infrastructure subsystem group (includes CGDF and PCF subsystems). Sixteen complex use cases describe the user interface subsystem. The engineering subsystem group is described by 14 average use cases and 8 simple use cases. And the infrastructure subsystem is described with 10 simple use cases. Therefore,

$$\text{UUCW} = (16 \text{ use cases} \times 15) + [(14 \text{ use cases} \times 10) + (8 \text{ use cases} \times 5)] + (10 \text{ use cases} \times 5) = 470$$

Analysis of the use cases indicates that there are 8 simple actors, 12 average actors, and 4 complex actors. Therefore,

$$\text{UAW} = (8 \text{ actors} \times 1) + (12 \text{ actors} \times 2) + 4 \text{ actors} \times 3) = 44$$

After evaluation of the technology and the environment,

$$\text{TCF} = 1.04$$

$$\text{ECF} = 0.96$$

Using relationship 33.2,

$$\text{UCP} = (470 + 44) \times 1.04 \times 0.96 = 513$$

Using past project data as a guide, the development group has produced 85 LOC per UCP. Therefore, an estimate of the overall size of the CAD project is 43,600 LOC. Similar computations can be made for applied effort or project duration.

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of $8,000 per month, the cost per line of code is approximately $13. Based on the use-case estimate and the historical productivity data, the total estimated project cost is $552,000 and the estimated effort is about 70 person-months.

Quote:

"Complicated methods might not yield a more accurate estimate, particularly when developers can incorporate their own intuition into the estimate."

Philip Johnson et al.

33.6.9 Reconciling Estimates

The estimation techniques discussed in the preceding sections result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost. The total estimated effort for the CAD software (section 33.6.3) ranges from a low of 46 person-months (derived using a process-based estimation approach) to a high of 68 person-months (derived with use-case estimation). The average estimate (using all four approaches) is 56 person-months. The variation from the average estimate is approximately 18 percent on the low side and 21 percent on the high side.

What happens when agreement between estimates is poor? The answer to this question requires a reevaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes: (1) the scope

of the project is not adequately understood or has been misinterpreted by the planner, or (2) productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied. You should determine the cause of divergence and then reconcile the estimates.

INFO

Automated Estimation Techniques for Software Projects

Automated estimation tools allow the planner to estimate cost and effort and to perform what-if analyses for important project variables such as delivery date or staffing. Although many automated estimation tools exist (see sidebar later in this chapter), all exhibit the same general characteristics and all perform the following six generic functions [Jon96]:

1. *Sizing of project deliverables.* The "size" of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), and descriptive information (e.g., documents).
2. *Selecting project activities.* The appropriate process framework is selected, and the software engineering task set is specified.
3. *Predicting staffing levels.* The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.
4. *Predicting software effort.* Estimation tools use one or more models (Section 33.7) that relate the size of the project deliverables to the effort required to produce them.
5. *Predicting software cost.* Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.
6. *Predicting software schedules.* When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities based on recommended models for effort distribution discussed later in this chapter.

When different estimation tools are applied to the same project data, a relatively large variation in estimated results can be encountered. More important, predicted values sometimes are significantly different than actual values. This reinforces the notion that the output of estimation tools should be used as one "data point" from which estimates are derived—not as the only source for an estimate.

33.7 Empirical Estimation Models

KEY POINT

An estimation model reflects the population of projects from which it has been derived. Therefore, the model is domain sensitive.

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.[9] Values for LOC or FP are estimated using the approach described in Sections 33.6.3 and 33.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously.

9 An empirical model using use cases as the independent variable is suggested in Section 33.6.6. However, relatively few have appeared in the literature to date.

An estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

33.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

$$E = A + B \times (e_v)^C \tag{33.3}$$

where A, B, and C are empirically derived constants, E is effort in person-months, and e_v is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (33.3), the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). A quick examination of any empirically derived model indicates that it must be calibrated for local needs.

33.7.2 The COCOMO II Model

In his classic book on software engineering economics, Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel*. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [Boe00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address different "stages" of the software process.

Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points,[10] function points, and lines of source code.

33.7.3 The Software Equation

The *software equation* [Put92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4,000 contemporary software projects. Based on these data, we derive an estimation model of the form

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4} \tag{33.4}$$

10 An *object point* is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application, along with complexity factors.

where

E = effort in person-months or person-years
t = project duration in months or years
B = "special skills factor"[11]
P = "productivity parameter" that reflects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

Typical values might be $P = 2{,}000$ for development of real-time embedded software, $P = 10{,}000$ for telecommunication and systems software, and $P = 28{,}000$ for business systems applications. The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

You should note that the software equation has two independent parameters: (1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [Put92] suggest a set of equations derived from the software equation. Minimum development time is defined as

$$t_{\min} = 8.14 \frac{\text{LOC}}{P^{0.43}} \text{ in months for } t_{\min} > 6 \text{ months} \tag{33.5a}$$

$$E = 180\, Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months} \tag{33.5b}$$

Note that t in Equation (33.5b) is represented in years.

Using Equation (33.5) with $P = 12{,}000$ (the recommended value for scientific software) for the CAD software discussed previously in this chapter,

$$t_{\min} = 8.14 \times \frac{33{,}200}{12{,}000^{0.43}} = 12.6 \text{ calendar months}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ person-months}$$

The results of the software equation correspond favorably with the estimates developed in Section 33.6. Like the COCOMO model noted in Section 33.7.2, the software equation continues to evolve. Further discussion of an extended version of this estimation approach can be found in [Put97b].

11 B increases slowly as "the need for integration, testing, quality assurance, documentation, and management skills grows" [Put92]. For small programs (KLOC = 5 to 15), $B = 0.16$. For programs greater than 70 KLOC, $B = 0.39$.

33.8 Estimation for Object-Oriented Projects

It is worthwhile to supplement conventional software cost estimation methods with a technique that has been designed explicitly for OO software. Lorenz and Kidd [Lor94] suggest the following approach:

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using the requirements model (Chapter 10), develop use cases and determine a count. Recognize that the number of use cases may change as the project progresses.
3. From the requirements model, determine the number of key classes (called analysis classes in Chapter 10).
4. Categorize the type of interface for the application and develop a multiplier for support classes, where the multipliers for no GUI, a text-based user interface, a conventional GUI, and a complex GUI are: 2.0, 2.25, 2.5, and 3.0, respectively. Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
5. Multiply the total number of classes (key + support) by the average number of work units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
6. Cross-check the class-based estimate by multiplying the average number of work units per use case.

33.9 Specialized Estimation Techniques

The estimation techniques discussed in Sections 33.6 through 33.8 can be used for any software project. However, when a software team encounters an extremely short project duration (weeks rather than months) that is likely to have a continuing stream of changes, project planning in general, and estimation in particular should be abbreviated.[12] In the sections that follow, I examine two specialized estimation techniques.

33.9.1 Estimation for Agile Development

Because the requirements for an agile project (Chapter 5) are defined by a set of user scenarios (e.g., "stories" in Extreme Programming), it is possible to develop an estimation approach that is informal, reasonably disciplined, and meaningful

12 "Abbreviated" does *not* mean eliminated. Even short-duration projects must be planned, and estimation is the foundation of solid planning.

within the context of project planning for each software increment. Estimation for agile projects uses a decomposition approach that encompasses the following steps:

? How are estimates developed when an agile process is applied?

1. Each user scenario (the equivalent of a mini use case created at the very start of a project by end users or other stakeholders) is considered separately for estimation purposes.
2. The scenario is decomposed into the set of software engineering tasks that will be required to develop it.

3a. Each task is estimated separately. Note: Estimation can be based on historical data, an empirical model, or "experience."

3b. Alternatively, the "volume" of the scenario can be estimated in LOC, FP, or some other volume-oriented measure (e.g., use case count).

4a. Estimates for each task are summed to create an estimate for the scenario.

4b. Alternatively, the volume estimate for the scenario is translated into effort using historical data.

5. The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

KEY POINT

In the context of estimation for agile projects, "volume" is an estimate of the overall size of a user scenario in LOC or FP.

Because the project duration required for the development of a software increment is quite short (typically three to six weeks), this estimation approach serves two purposes: (1) to be certain that the number of scenarios to be included in the increment conforms to the available resources, and (2) to establish a basis for allocating effort as the increment is developed.

33.9.2 Estimation for WebApp Projects

WebApp projects often adopt the agile process model. A modified function point measure, coupled with the steps outlined in Section 33.9.1, can be used to develop an estimate for the WebApp. Roetzheim [Roe00] suggests the following approach when adapting function points for WebApp estimation:

- *Inputs* are each input screen or form (for example, CGI or Java), each maintenance screen, and if you use a tab notebook metaphor anywhere, each tab.
- *Outputs* are each static Web page, each dynamic Web page script (for example, ASP, ISAPI, or other DHTML script), and each report (whether Web based or administrative in nature).
- *Tables* are each logical table in the database plus, if you are using XML to store data in a file, each XML object (or collection of XML attributes).
- *Interfaces* retain their definition as logical files (for example, unique record formats) into our out-of-the-system boundaries.

- *Queries* are each externally published or use a message-oriented interface. A typical example is DCOM or COM external references.

Function points (interpreted in the manner noted) are a reasonable indicator of volume for a WebApp.

SOFTWARE TOOLS

Effort and Cost Estimation

Objective: The objective of effort and cost estimation tools is to provide a project team with estimates of effort required, project duration, and cost in a manner that addresses the specific characteristics of the project at hand and the environment in which the project is to be built.

Mechanics: In general, cost estimation tools make use of an historical database derived from local projects and data collected across the industry, and an empirical model (e.g., COCOMO II) that is used to derive effort, duration, and cost estimates. Characteristics of the project and the development environment are input and the tool provides a range of estimation outputs.

Representative Tools:[13]

Costar, developed by Softstar Systems (**www.softstarsystems.com**), uses the COCOMO II model to develop software estimates.

Cost Xpert, developed by Cost Xpert Group (**www.costxpert.com**), integrates multiple estimation models and an historical project database.

Construx Professional, developed by Construx (**http://www.construx.com/Resources/Construx_Estimate/**), is based on Putnam Model and COCOMO II.

Knowledge Plan, developed by Software Productivity Research (**www.spr.com**), uses function point input as the primary driver for a complete estimation package.

Price S, developed by Price Systems (**www.pricesystems.com**), is one of the oldest and most widely used estimating tools for large-scale software development projects.

SEER/SEM, developed by Galorath (**www.galorath.com**), provides comprehensive estimation capability, sensitivity analysis, risk assessment, and other features.

SLIM-Estimate, developed by QSM (**www.qsm.com**), draws on comprehensive "industry knowledge bases" to provide a "sanity check" for estimates derived using local data.

33.10 THE MAKE/BUY DECISION

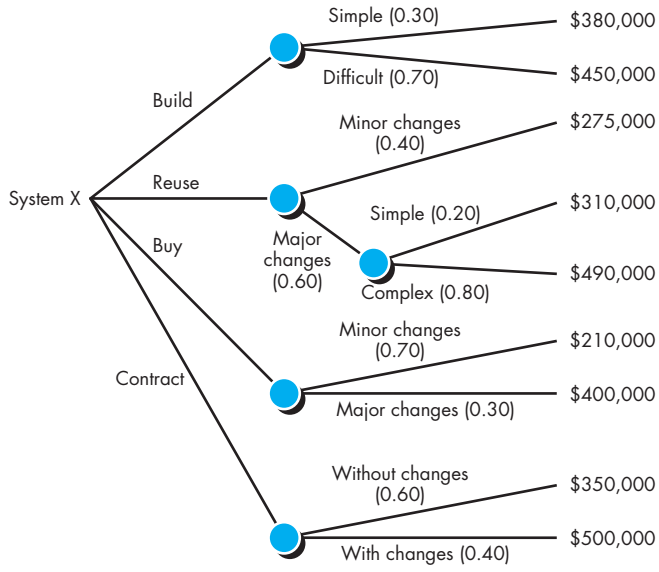
In many software application areas, it is often more cost-effective to acquire than develop computer software. Software engineering managers are faced with a make/buy decision that can be further complicated by a number of acquisition options: (1) software may be purchased (or licensed) off-the-shelf, (2) "full-experience" or "partial-experience" software components (see Section 33.4.2) may be acquired and then modified and integrated to meet specific needs, or (3) software may be custom built by an outside contractor to meet the purchaser's specifications.

The steps involved in the acquisition of software are defined by the criticality of the software to be purchased and the end cost. In some cases (e.g., low-cost

13 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

FIGURE 33.5

A decision tree to support the make/buy decision



apps), it is less expensive to purchase and experiment than to conduct a lengthy evaluation of potential software packages. In the final analysis, the make/buy decision is made based on the following conditions: (1) Will the delivery date of the software product be sooner than that for internally developed software? (2) Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally? (3) Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support? These conditions apply for each of the acquisition options.

33.10.1 Creating a Decision Tree

? Is there a systematic way to sort through the options associated with the make/buy decision?

The steps just described can be augmented using statistical techniques such as decision tree analysis.[14] For example, Figure 33.5 depicts a decision tree for a software-based system X. In this case, the software engineering organization can (1) build system X from scratch, (2) reuse existing partial-experience components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.

If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. Using the estimation techniques discussed previously in this chapter, the project planner projects that a difficult development effort will

14 A worthwhile introduction to decision tree analysis can be found at **http://en.wikipedia.org/wiki/Decision_tree**.

cost $450,000. A "simple" development effort is estimated to cost $380,000. The expected value for cost, computed along any branch of the decision tree, is

$$\text{Expected cost} = \Sigma\ (\text{path probability})_i \times (\text{estimated path cost})_i$$

where i is the decision tree path. For the build path,

$$\text{Expected cost}_{\text{build}} = 0.30\ (\$380\text{K}) + 0.70\ (\$450\text{K}) = \$429\text{K}$$

Following other paths of the decision tree, the projected costs for reuse, purchase, and contract, under a variety of circumstances, are also shown. The expected costs for these paths are

$$\text{Expected cost}_{\text{reuse}} = 0.40\ (\$275\text{K}) + 0.60\ [0.20\ (\$310\text{K}) + 0.80\ (\$490\text{K})] = \$382\text{K}$$

$$\text{Expected cost}_{\text{buy}} = 0.70\ (\$210\text{K}) + 0.30\ (\$400\text{K}) = \$267\text{K}$$

$$\text{Expected cost}_{\text{contract}} = 0.60\ (\$350\text{K}) + 0.40\ (\$500\text{K}) = \$410\text{K}$$

Based on the probability and projected costs that have been noted in Figure 33.5, the lowest expected cost is the "buy" option.

It is important to note, however, that many criteria—not just cost— must be considered during the decision-making process. Availability, experience of the developer/vendor/contractor, conformance to requirements, local "politics," and the likelihood of change are but a few of the criteria that may affect the ultimate decision to build, reuse, buy, or contract.

33.10.2 Outsourcing

Sooner or later, every company that develops computer software asks a fundamental question: "Is there a way that we can get the software and systems we need at a lower price?" The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: *outsourcing*.

Quote:

"As a rule outsourcing requires even more skillful management than in-house development."

Steve McConnell

In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.[15]

The decision to outsource can be either strategic or tactical. At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others. At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.

15 Outsourcing can be viewed more generally as any activity that leads to the acquisition of software or software components from a source outside the software engineering organization.

Regardless of the breadth of focus, the outsourcing decision is often a financial one. A detailed discussion of the financial analysis for outsourcing is beyond the scope of this book and is best left to others (e.g., [Min95]). However, a brief review of the pros and cons of the decision is worthwhile.

On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them. On the negative side, a company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

The trend toward outsourcing will undoubtedly continue. The only way to blunt the trend is to recognize that software work is extremely competitive at all levels. The only way to survive is to become as competitive as the outsourcing vendors themselves.

SafeHome

Outsourcing

The scene: Meeting room at CPI Corporation early in the project.

The players: Mal Golden, senior manager, product development; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development; and Doug Miller, project manager, software engineering.

The conversation:

Joe: We're considering outsourcing the *SafeHome* software engineering portion of the product.

Doug (shocked): When did this happen?

Lee: We got a quote from an offshore developer. It comes in at 30 percent below what your group seems to believe it will cost. Here.

[Hands the quote to Doug who reads it.]

Mal: As you know, Doug, we're trying to keep costs down and 30 percent is 30 percent. Besides, these people come highly recommended.

Doug (taking a breath and trying to remain calm): You guys caught me by surprise here, but before you make a final decision a few comments?

Joe (nodding): Sure, go ahead.

Doug: We haven't worked with this outsourcing company before, right?

Mal: Right, but . . .

Doug: And they note that any changes to spec will be billed at an additional rate, right?

Joe (frowning): True, but we expect that things will be reasonably stable.

Doug: A bad assumption, Joe.

Joe: Well, . . .

Doug: It's likely that we'll release new versions of this product over the next few years. And it's reasonable to assume that software will provide many of the new features, right?

[All nod.]

Doug: Have we ever coordinated an international project before?

Lee (looking concerned): No, but I'm told . . .

Doug (trying to suppress his anger): So what you're telling me is: (1) we're about to work with an unknown vendor, (2) the costs to do this are not as low as they seem, (3) we're de facto committing to work with them over many product releases, no matter what they do on the first one, and (4) we're going to learn on the job relative to an international project.

[All remain silent.]

Doug: Guys . . . I think this is a mistake, and I'd like you to take a day to reconsider. We'll have far more control if we do the work in-house. We have the expertise, and I can guarantee that it won't cost us much more . . . the risk will be lower, and I know you're all risk averse, as I am.

Joe (frowning): You've made a few good points, but you have a vested interest in keeping this project in-house.

Doug: That's true, but it doesn't change the facts.

Joe (with a sigh): Okay, let's table this for a day or two, give it some more thought, and meet again for a final decision. Doug, can I speak with you privately?

Doug: Sure . . . I really do want to be sure we do the right thing.

33.11 Summary

A software project planner must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved. In addition, the planner must predict the resources (hardware and software) that will be required and the risk involved.

The statement of scope helps the planner to develop estimates using one or more techniques that fall into two broad categories: decomposition and empirical modeling. Decomposition techniques require a delineation of major software functions, followed by estimates of either (1) the number of LOC, (2) selected values within the information domain, (3) the number of use cases, (4) the number of person-months required to implement each function, or (5) the number of person-months required for each software engineering activity. Empirical techniques use empirically derived expressions for effort and time to predict these project quantities. Automated tools can be used to implement a specific empirical model.

Accurate project estimates generally use at least two of the three techniques just noted. By comparing and reconciling estimates developed using different techniques, the planner is more likely to derive an accurate estimate. Software project estimation can never be an exact science, but a combination of good historical data and systematic techniques can improve estimation accuracy.

Problems and Points to Ponder

33.1. Assume that you are the project manager for a company that builds software for household robots. You have been contracted to build the software for a robot that mows the lawn for a homeowner. Write a statement of scope that describes the software. Be sure your statement of scope is bounded. If you're unfamiliar with robots, do a bit of research before you begin writing. Also, state your assumptions about the hardware that will be required. Alternate: Replace the lawn-mowing robot with another problem that is of interest to you.

33.2. Software project complexity is discussed briefly in Section 33.1. Develop a list of software characteristics (e.g., concurrent operation, graphical output) that affect the complexity of a project. Prioritize the list.

33.3. Performance is an important consideration during planning. Discuss how performance can be interpreted differently depending upon the software application area.

33.4. Do a functional decomposition of the robot software you described in Problem 33.1. Estimate the size of each function in LOC. Assuming that your organization produces 450 LOC/pm with a burdened labor rate of $7,000 per person-month, estimate the effort and cost required to build the software using the LOC-based estimation technique described in this chapter.

33.5. Use the software equation to estimate the lawn-mowing robot software. Assume that Equation (33.4) is applicable and that P = 8,000.

33.6. Develop a spreadsheet model that implements one or more of the estimation techniques described in this chapter. Alternatively, acquire one or more online models for estimation from Web-based sources.

33.7. For a project team: Develop a software tool that implements each of the estimation techniques developed in this chapter.

33.8. It seems odd that cost and schedule estimates are developed during software project planning—before detailed software requirements analysis or design has been conducted. Why do you think this is done? Are there circumstances when it should not be done?

Further Readings and Information Sources

Most software project management books contain discussions of project estimation. The Project Management Institute (*PMBOK Guide,* PMI, 2001), Wysoki (*Effective Project Management: Traditional, Agile, Extreme,* 6th ed., Wiley, 2011), Lewis (*Project Planning Scheduling and Control,* 5th ed., McGraw-Hill, 2010), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 10th ed., Wiley, 2009), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3rd ed., Wiley, 2000), and Phillips [Phi98] provide useful estimation guidelines.

McConnell (*Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006) has written a pragmatic guide that provides worthwhile guidance for anyone who must estimate the cost of software. Parthasarathy (*Practical Software Estimation,* Addison-Wesley, 2007) emphasizes function points as an estimation metric. Hill (*Practical Software Project Estimation,* McGraw-Hill Osborne Media, 2010) and Laird and Brennan (*Software Measurement and Estimation: A Practical Approach,* Wiley-IEEE Computer Society Press, 2006) addresses measurement and its use in software estimation. Pfleeger (*Software Cost Estimation and Sizing Methods, Issues, and Guidelines,* RAND Corporation, 2005) has developed an abbreviated guidebook that addresses many estimation fundamentals. Jones (*Estimating Software Costs,* 2nd ed., McGraw-Hill, 2007) has written one of the most comprehensive treatments of models and data that are applicable to software estimating in every application domain. Coombs (*IT Project Estimation,* Cambridge University Press, 2002, and Roetzheim and Beasley (*Software Project Cost and Schedule Estimating: Best Practices,* Prentice Hall, 1997) present many useful models and suggest step-by-step guidelines for generating the best possible estimates.

A wide variety of information sources on software estimation is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

34 Project Scheduling

Key Concepts

critical path. 757
earned value. 772
effort distribution. . 760
people and effort . 759
scheduling
principles of . . . 758
WebApp and mobile projects 769
task network 764
time-boxing 768
time-line charts. . . 766
tracking 767
work breakdown . 765

In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking.

His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months.

He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going.

"Really great," said the young engineer with youthful enthusiasm. "This is much simpler than I thought. I'm probably close to 75 percent finished."

Quick Look

What is it? You've selected an appropriate process model, you've identified the software engineering tasks that have to be performed, you estimated the amount of work and the number of people, you know the deadline, you've even considered the risks. Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time. Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality. In a nutshell, that's software project scheduling and tracking.

Who does it? At the project level, software project managers using information solicited from software engineers. At an individual level, software engineers themselves.

Why is it important? In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

What are the steps? The software engineering tasks dictated by the software process model are refined for the functionality to be built. Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established.

What is the work product? The project schedule and related information are produced.

How do I ensure that I've done it right? Proper scheduling requires that: (1) all tasks appear in the network, (2) effort and timing are intelligently allocated to each task, (3) interdependencies between tasks are properly indicated, (4) resources are allocated for the work to be done, and (5) closely spaced milestones are provided so that progress can be tracked.

The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time.

A week later the boss called the engineer into his office and asked, "Where are we?"

"Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon."

"How does the deadline look?" the boss asked.

"No problem," said the engineer. "I'm close to 90 percent complete."

If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer[1] stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late.

This story has been repeated tens of thousands of times by software developers during the past five decades. The big question is why?

34.1 Basic Concepts

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes:

- An unrealistic deadline established by someone outside the software team and forced on managers and practitioners on the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Quote:

"Excessive or irrational schedules are probably the single most destructive influence in all of software."

Capers Jones

Aggressive (read "unrealistic") deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate, from the point of view of the person who sets the deadline. But common sense says that legitimacy must also be perceived by the people doing the work.

1. In case you were wondering, this story is autobiographical (RSP).

The estimation methods discussed in Chapter 33 and the scheduling techniques described in this chapter are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should "protect his or her team from undue [schedule] pressure . . . [and] reflect the pressure back to its originators" [Pag85].

Quote:

"I love deadlines. I like the whooshing sound they make as they fly by."

Douglas Adams

To illustrate, assume that your software team has been asked to build a real-time controller for a medical diagnostic instrument that is to be introduced to the market in nine months. After careful estimation and risk analysis (Chapter 35), you come to the conclusion that the software, as requested, will require 14 calendar months to create with available staff. How should you proceed?

It is unrealistic to march into the customer's office (in this case the likely customer is marketing/sales) and demand that the delivery date be changed. External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work (from a career standpoint). So, what to do? we recommend the following steps in this situation:

? What should we do when management demands that we make a deadline that is impossible?

1. Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model (Chapter 4), develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
3. Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percent improvement that would be required to achieve the deadline as it currently exists.[2] The following comment is appropriate:

   > I think we may have a problem with the delivery date for the XYZ controller software. I've given each of you an abbreviated breakdown of development rates for past software projects and an estimate that we've done a number of different ways. You'll note that I've assumed a 20 percent improvement in past development rates, but we still get a delivery date that's 14 calendar months rather than 9 months away.

4. Offer the incremental development strategy as an alternative:

   > We have a few options, and I'd like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we'll have a shot at getting this job done in nine months. But understand that this will increase the risk of poor quality due to the tight time line.[3] Second, we can remove a number of

2. If the required improvement is 10 to 25 percent, it may actually be possible to get the job done. But, more likely, the required improvement in team performance will be greater than 50 percent. This is an unrealistic expectation.
3. You might also add that increasing the number of people does not reduce calendar time proportionally.

> the software functions and capabilities that you're requesting. This will make the preliminary version of the product somewhat less functional, but we can announce all functionality and then deliver over the 14-month period. Third, we can dispense with reality and wish the project complete in nine months. We'll wind up with nothing that can be delivered to a customer. The third option, I hope you'll agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster.

There will be some grumbling, but if solid estimates based on good historical data are presented, it's likely that negotiated versions of option 1 or 2 will be chosen. The unrealistic deadline evaporates.

34.2 Project Scheduling

Fred Brooks was once asked how software projects fall behind schedule. His response was as simple as it was profound: "One day at a time."

The reality of a technical project (whether it involves building a hydroelectric plant or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the *critical path*. If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

ADVICE

The tasks required to achieve a project manager's objective should not be performed manually. There are many excellent scheduling tools. Use them.

As a project manager, your objective is to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress to ensure that delay is recognized "one day at a time." To accomplish this, you must have a schedule that has been defined at a degree of resolution that allows progress to be monitored and the project to be controlled.

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled.

Quote:

"Overly optimistic scheduling doesn't result in shorter actual schedules, it results in longer ones."

Steve McConnell

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering

organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

34.2.1 Basic Principles

KEY POINT

When you develop a schedule, compartmentalize the work, note task interdependencies, allocate effort and time to each task, and define responsibilities, outcomes, and milestones.

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort).[4] On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 19) and has been approved.

Each of these principles is applied as the project schedule evolves.

4. In reality, less than three person-days of effort are available because of unrelated meetings, sickness, vacation, and a variety of other reasons. For our purposes, however, we assume 100 percent availability.

34.2.2 The Relationship between People and Effort

ADVICE

If you must add people to a late project, be sure that you've assigned them work that is highly compartmentalized.

There is a common myth that is still believed by many managers who are responsible for software development work: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.

In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

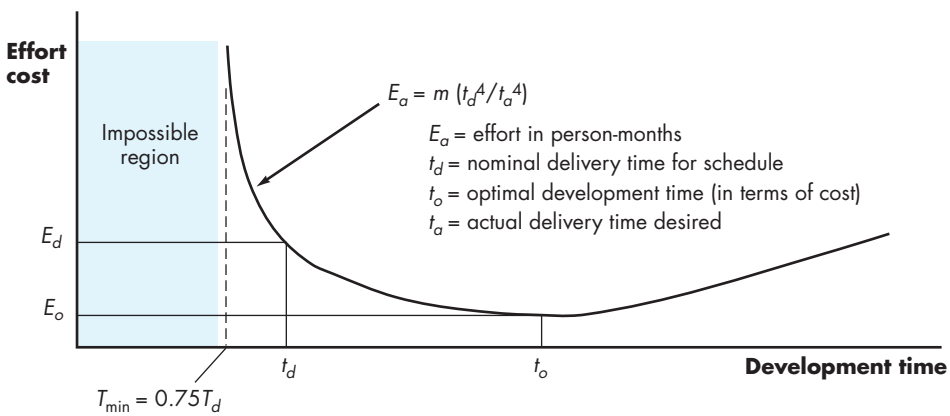
Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources).

The *Putnam-Norden-Rayleigh (PNR) Curve*[5] provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 34.1. The curve indicates a minimum value t_o that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of t_o (i.e., as we try to accelerate delivery), the curve rises nonlinearly.

As an example, we assume that a project team has estimated a level of effort E_d will be required to achieve a nominal delivery time t_d that is optimal in terms

FIGURE 34.1

The relationship between effort and delivery time



5. Original research can be found in [Nor70] and [Put78].

KEY POINT

If delivery can be delayed, the PNR curve indicates that project costs can be reduced substantially.

of schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of t_d. In fact, the PNR curve indicates the project delivery time cannot be compressed much beyond $0.75t_d$. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option, $t_o = 2t_d$. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.

The software equation [Put92] introduced in Chapter 33 is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), L, is related to effort and development time by the equation:

ADVICE

As the project deadline becomes tighter and tighter, you reach a point at which the work cannot be completed on schedule, regardless of the number of people doing the work. Face reality and define a new delivery date.

$$L = P \times E^{1/3} t^{4/3}$$

where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that leads to high-quality software engineering work (typical values for P range between 2000 and 12,000), and t is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort E:

$$E = \frac{L^3}{P^3 t^4} \tag{34.1}$$

where E is the effort expended (in person-years) over the entire life cycle for software development and maintenance and t is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (34.1) yields:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ person-years}$$

This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

34.2.3 Effort Distribution

Each of the software project estimation techniques discussed in Chapter 33 leads to estimates of work units (e.g., person-months) required to complete software

? How should effort be distributed across the software process workflow?

development. A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule*. Forty percent of all effort is allocated to front-end analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.

This effort distribution should be used as a guideline only.[6] The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

34.3 Defining a Task Set for the Software Project

Regardless of the process model that is chosen, the work that a software team performs is achieved through a set of tasks that enable you to define, develop, and ultimately support computer software. No single task set is appropriate for all projects. The set of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product. Therefore, an effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

As we noted in Chapter 3, a task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project. The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.

In order to develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work. Although it

6. Today, the 40-20-40 rule is under attack. Some believe that more than 40 percent of overall effort should be expended during analysis and design. On the other hand, some proponents of agile development (Chapter 5) argue that less time should be expended "up front" and that a team should move quickly to construction.

is difficult to develop a comprehensive taxonomy of software project types, most software organizations encounter the following projects:

1. *Concept development projects* that are initiated to explore some new business concept or application of some new technology.
2. *New application development* projects that are undertaken as a consequence of a specific customer request.
3. *Application enhancement* projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
4. *Application maintenance projects* that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
5. *Reengineering projects* that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

WebRef

An adaptable process model (APM) has been developed to assist in defining task sets for various software projects. A complete description of the APM can be found at **www.rspa.com/apm.**

Even within a single project type, many factors influence the task set to be chosen. These include [Pre05]: size of the project, number of potential users, mission criticality, application longevity, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, embedded and nonembedded characteristics, project staff, and reengineering factors. When taken in combination, these factors provide an indication of the *degree of rigor* with which the software process should be applied.

34.3.1 A Task Set Example

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following major tasks:

1.1 **Concept scoping** determines the overall scope of the project.

1.2 **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.

1.3 **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.

1.4 **Proof of concept** demonstrates the viability of a new technology in the software context.

1.5 **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

1.6 **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these tasks should yield few surprises. In fact, the software engineering flow for concept development projects (and for all other types of projects as well) is little more than common sense.

34.3.2 Refinement of Major Tasks

The major tasks (i.e., software engineering actions) described in the preceding section may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each major task and decomposing it into a set of subtasks (with related work products and milestones).

As an example of task decomposition, consider Task 1.1, Concept Scoping. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping activity:

```
Task definition: Task 1.1 Concept Scoping
1.1.1   Identify need, benefits and potential customers;
1.1.2   Define desired output/control and input events that drive the application;
        Begin Task 1.1.2
        1.1.2.1 TR: Review written description of need[7]
        1.1.2.2 Derive a list of customer visible outputs/inputs
        1.1.2.3 TR: Review outputs/inputs with customer and revise as required;
        endtask Task 1.1.2
1.1.3   Define the functionality/behavior for each major function;
        Begin Task 1.1.3
        1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
        1.1.3.2 Derive a model of functions/behaviors;
        1.1.3.3 TR: Review functions/behaviors with customer and revise as required;
        endtask Task 1.1.3
1.1.4   Isolate those elements of the technology to be implemented in software;
1.1.5   Research availability of existing software;
1.1.6   Define technical feasibility;
1.1.7   Make quick estimate of size;
1.1.8   Create a Scope Definition;
endtask definition: Task 1.1
```

The tasks and subtasks noted in the process design language refinement form the basis for a detailed schedule for the concept scoping activity.

7. TR indicates that a technical review (Chapter 20) is to be conducted.

34.4 Defining a Task Network

Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

KEY POINT

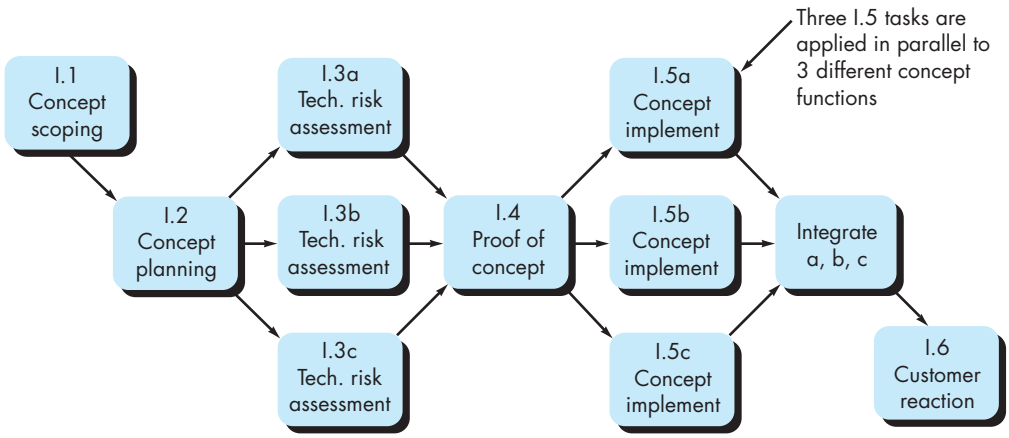
The task network is a useful mechanism for depicting intertask dependencies and determining the critical path.

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering tasks. Figure 34.2 shows a schematic task network for a concept development project.

The concurrent nature of software engineering activities leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, you should determine intertask dependencies to ensure continuous progress toward completion. In addition, you should be aware of those tasks that lie on the *critical path.* That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule. These issues are discussed in more detail later in this chapter.

It is important to note that the task network shown in Figure 34.2 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each activity shown in the figure would be expanded. For example, Task 1.1 would be expanded to show all tasks detailed in the refinement of Tasks 1.1 shown in Section 34.3.2.

Figure 34.2 A task network for concept development



34.5 Scheduling

Quote:

"*All* we have to decide is what to do with the time that is given to us."

Gandalf in The Lord of the Rings: Fellowship of the Rings

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

Program evaluation and review technique (PERT) and the *critical path method* (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected.

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish "most likely" time estimates for individual tasks by applying statistical models, and (3) calculate "boundary times" that define a time "window" for a particular task.

Software Tools

Project Scheduling

Objective: The objective of project scheduling tools is to enable a project manager to define work tasks; establish their dependencies; assign human resources to tasks; and develop a variety of graphs, charts, and tables that aid in tracking and control of the software project.

Mechanics: In general, project scheduling tools require the specification of a work breakdown structure of tasks or the generation of a task network. Once the task breakdown (an outline) or network is defined, start and end dates, human resources, hard deadlines, and other data are attached to each. The tool then generates a variety of time-line charts and other tables that enable a manager to assess the task flow of a project. These data can be updated continually as the project is under way.

Representative Tools:[8]

AMS Realtime, developed by Advanced Management Systems (**www.amsusa.com**), provides scheduling capabilities for projects of all sizes and types.

Microsoft Project, developed by Microsoft (**www.microsoft.com**), is the most widely used PC-based project scheduling tool.

4C, developed by *4C Systems* (**www.4csys.com**), supports all aspects of project planning including scheduling.

A comprehensive list of project management software vendors and products can be found at **www.infogoal.com/pmc/pmcswr.htm**.

8. Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

34.5.1 Time-Line Charts

When creating a software project schedule, you begin with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.

KEY POINT

A time-line chart enables you to determine what tasks will be conducted at a given point in time.

As a consequence of this input, a *time-line chart,* also called a *Gantt chart,* is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Figure 34.3 illustrates the format of a time-line chart. It depicts a part of a software project schedule that emphasizes the **concept scoping** task for a word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce *project*

FIGURE 34.3 An example time-line chart

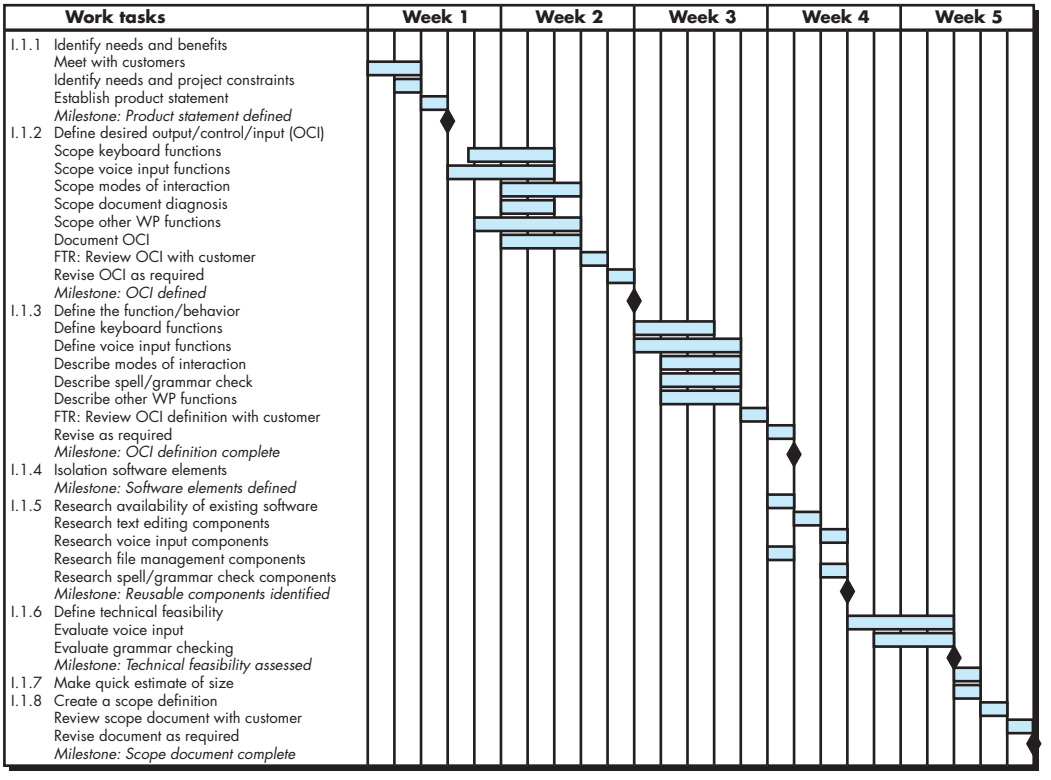


FIGURE 34.4 An example project table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	require more
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	effort/time
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

tables—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information (Figure 34.4). Used in conjunction with the time-line chart, project tables enable you to track progress.

34.5.2 Tracking the Schedule

Quote:

"The basic rule of software status reporting can be summarized in a single phrase: 'No surprises'."

Capers Jones

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones (the diamonds shown in Figure 34.3) have been accomplished by the scheduled date.
- Comparing the actual start date to the planned start date for each project task listed in the resource table (Figure 34.4).
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis (Section 34.6) to assess progress quantitatively.

In reality, all of these tracking techniques are used by experienced project managers.

ADVICE

The best indication of progress is the completion and successful review of a defined software work product.

Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur, you must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [Jal04]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm (Chapter 4) is chosen, and a schedule is derived for each incremental delivery.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative: "If the work isn't finished, how can we proceed?" The answer lies in the way work is accomplished. By the time the time-box boundary is encountered, it is likely that 90 percent of the task has been completed.[9] The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

34.5.3 Tracking Progress for an OO Project

Although an iterative model is the best framework for an OO project, task parallelism makes project tracking difficult. You may have difficulty establishing meaningful milestones for an OO project because a number of different things are happening at once. In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed

- All classes and the class hierarchy have been defined and reviewed.
- Class attributes and operations associated with a class have been defined and reviewed.
- Class relationships (Chapter 10) have been established and reviewed.

9. A cynic might recall the saying: "The first 90 percent of the system takes 90 percent of the time; the remaining 10 percent of the system takes 90 percent of the time."

- A behavioral model (Chapter 11) has been created and reviewed.
- Reusable classes have been noted.

Technical milestone: OO design completed

- The set of subsystems has been defined and reviewed.
- Classes are allocated to subsystems and reviewed.
- Task allocation has been established and reviewed.
- Responsibilities and collaborations have been identified.
- Attributes and operations have been designed and reviewed.
- The communication model has been created and reviewed.

Technical milestone: OO programming completed

- Each new class has been implemented in code from the design model.
- Extracted classes (from a reuse library) have been implemented.
- Prototype or increment has been built.

Technical milestone: OO testing

ADVICE

Debugging and testing occur in concert with one another. The status of debugging is often assessed by considering the type and number of "open" errors (bugs).

- The correctness and completeness of OO analysis and design models have been reviewed.
- A class-responsibility-collaboration network (Chapter 10) has been developed and reviewed.
- Test cases are designed, and class-level tests (Chapter 24) have been conducted for each class.
- Test cases are designed, and cluster testing (Chapter 24) is completed and the classes are integrated.
- System-level tests have been completed.

Recalling that the OO process model is iterative, each of these milestones may be revisited as different increments are delivered to the customer.

34.5.4 Scheduling for WebApp and Mobile Projects

Web and MobileApp project scheduling distributes estimated effort across the planned time line (duration) for building each increment. This is accomplished by allocating the effort to specific tasks. It is important to note, however, that the overall schedule evolves over time. During the first iteration, a macroscopic schedule is developed. This type of schedule identifies all Web or MobileApp increments and projects the dates on which each will be deployed. As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule. Here, specific development tasks (required to accomplish an activity) are identified and scheduled.

Let's consider the **SafeHomeAssured.com** to better understand macroscopic scheduling. Recalling previous discussions of **SafeHomeAssured.com**, seven increments can be identified for the Web-based component of the project:

Increment 1: Basic company and product information

Increment 2: Detailed product information and downloads

Increment 3: Product quotes and processing product orders

Increment 4: Space layout and security system design

Increment 5: Information and ordering of monitoring services

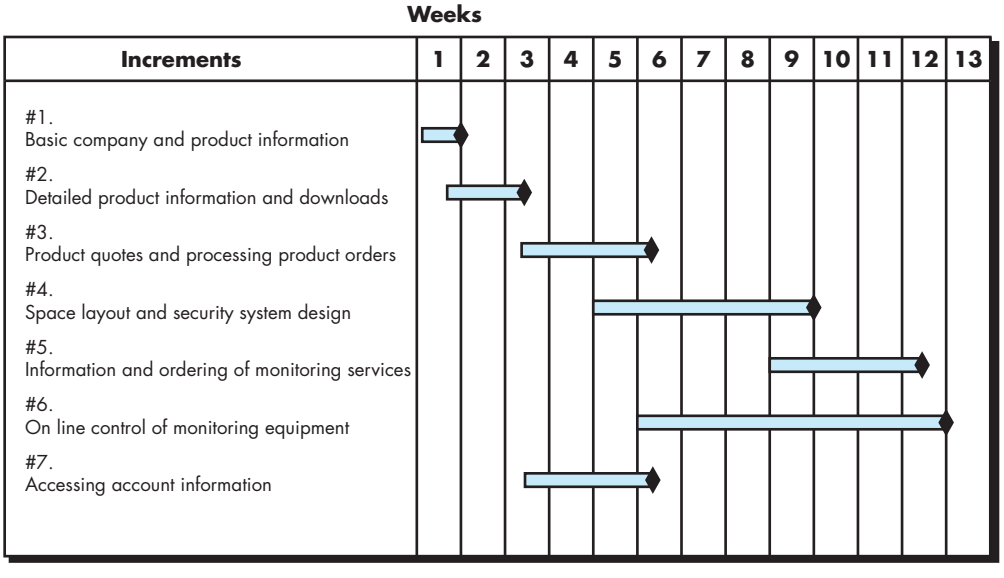
Increment 6: Online control of monitoring equipment

Increment 7: Accessing account information

The team consults and negotiates with stakeholders and develops a *preliminary* deployment schedule for all seven increments. A time-line chart for this schedule is illustrated in Figure 34.5.

It is important to note that the deployment dates (represented by diamonds on the time-line chart) are preliminary and may change as more detailed scheduling of the increments occurs. However, this macroscopic schedule provides management with an indication of when content and functionality will be available and when the entire project will be completed. As a preliminary estimate, the team will work to deploy all increments with a 12-week time line. It's also worth noting that some of the increments will be developed in parallel (e.g., increments 3, 4, and 7). This assumes that the team will have sufficient people to do this parallel work.

FIGURE 34.5 Time line for macroscopic project schedule



Once the macroscopic schedule has been developed, the team is ready to schedule work tasks for a specific increment. To accomplish this, you can use a generic process framework that is applicable for all increments. A *task list* is created by using the generic tasks derived as part of the framework as a starting point and then adapting these by considering the content and functions to be derived for a specific WebApp increment.

Each framework action (and its related tasks) can be adapted in one of four ways: (1) a task is applied as is, (2) a task is eliminated because it is not necessary for the increment, (3) a new (custom) task is added, and (4) a task is refined (elaborated) into a number of named subtasks that each becomes part of the schedule.

To illustrate, consider a generic *design modeling* action for WebApps that can be accomplished by noting the generic design tasks for WebApps discussed in Chapter 17. As an example, consider the generic task *Design the Interface* as it is applied to the fourth increment of **SafeHomeAssured.com**. Recall that the fourth increment implements the content and function for describing the living or business space to be secured by the *SafeHome* security system. Referring to Figure 34.5, the fourth increment commences at the beginning of the fifth week and terminates at the end of the ninth week.

There is little question that the *Design the Interface* task must be conducted. The team recognizes that the interface design is pivotal to the success of the increment and decides to refine (elaborate) the task. The following subtasks are derived for the *Design the Interface* task for the fourth increment:

- Develop a sketch of the page layout for the space design page.
- Review layout with stakeholders.
- Design space layout navigation mechanisms.
- Design "drawing board" layout.[10]
- Develop procedural details for the graphical wall layout function.
- Develop procedural details for the wall length computation and display function.
- Develop procedural details for the graphical window layout function.
- Develop procedural details for the graphical door layout function.
- Design mechanisms for selecting security system components (sensors, cameras, microphones, etc.).

10. At this stage, the team envisions creating the space by literally drawing the walls, windows, and doors using graphical functions. Wall lines will "snap" onto grip points. Dimensions of the wall will be displayed automatically. Windows and doors will be positioned graphically. The end user can also select specific sensors, cameras, etc., and position them once the space has been defined.

- Develop procedural details for the graphical layout of security system components.
- Conduct pair walkthroughs as required.

These tasks become part of the increment schedule for the fourth WebApp increment and are allocated over the increment development schedule. They can be input to scheduling software (e.g., Microsoft *Project*) and used for tracking and control.

SAFEHOME

Tracking the Schedule

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug (glancing at a PowerPoint slide): The schedule for the first *SafeHome* increment seems reasonable, but we're going to have trouble tracking progress.

Vinod (a concerned look on his face): Why? We have tasks scheduled on a daily basis, plenty of work products, and we've been sure that we're not overallocating resources.

Doug: All good, but how do we know when the requirements model for the first increment is complete?

Jamie: Things are iterative, so that's difficult.

Doug: I understand that, but . . . well, for instance, take "analysis classes defined." You indicated that as a milestone.

Vinod: We have.

Doug: Who makes that determination?

Jamie (aggravated): They're done when they're done.

Doug: That's not good enough, Jamie. We have to schedule TRs [technical reviews, Chapter 20], and you haven't done that. The successful completion of a review on the analysis model, for instance, is a reasonable milestone. Understand?

Jamie (frowning): Okay, back to the drawing board.

Doug: It shouldn't take more than an hour to make the corrections . . . everyone else can get started now.

34.6 EARNED VALUE ANALYSIS

KEY POINT

Earned value provides a quantitative indication of progress.

We discussed a number of qualitative approaches to project tracking in Section 34.5. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA). Humphrey [Hum95] discusses earned value in the following manner:

> The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

Stated even more simply, earned value is a measure of progress. It enables you to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [Fle98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project." To determine the earned value, the following steps are performed:

? How do I compute earned value and use it to assess progress?

1. The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, $BCWS_i$ is the effort planned for work task *i*. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the $BCWS_i$ values for all work tasks that should have been completed by that point in time on the project schedule.
2. The BCWS values for all work tasks are summed to derive the *budget at completion* (BAC). Hence,

   $$BAC = \Sigma\ (BCWS_k) \text{ for all tasks } k$$

3. Next, the value for *budgeted cost of work performed* (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

WebRef
A wide array of earned value analysis resources can be found at **http://www.acq.osd.mil/evm/.**

Wilkens [Wil99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed." Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

$$\text{Schedule performance index, SPI} = \frac{BCWP}{BCWS}$$

$$\text{Schedule variance, SV} = BCWP - BCWS$$

SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

$$\text{Percent scheduled for completion} = \frac{BCWS}{BAC}$$

provides an indication of the percentage of work that should have been completed by time *t*.

$$\text{Percent complete} = \frac{BCWP}{BAC}$$

provides a quantitative indication of the percent of completeness of the project at a given point in time *t*.

It is also possible to compute the *actual cost of work performed* (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

$$\text{Cost performance index, CPI} = \frac{\text{BCWP}}{\text{ACWP}}$$

$$\text{Cost variance, CV} = \text{BCWP} - \text{ACWP}$$

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

34.7 Summary

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical path, a time-line chart, and a variety of project information. Using the schedule as a guide, you can track and control each step in the software process.

Problems and Points to Ponder

34.1. "Unreasonable" deadlines are a fact of life in the software business. How should you proceed if you're faced with one?

34.2. What is the difference between a macroscopic schedule and a detailed schedule? Is it possible to manage a project if only a macroscopic schedule is developed? Why?

34.3. Is there ever a case where a software project milestone is not tied to a review? If so, provide one or more examples.

34.4. "Communication overhead" can occur when multiple people work on a software project. The time spent communicating with others reduces individual productively (LOC/month), and the result can be less productivity for the team. Illustrate (quantitatively) how engineers who are well versed in good software engineering practices and use technical reviews can increase the production rate of a team (when compared to the sum of individual production rates). Hint: You can assume that reviews reduce rework and that rework can account for 20 to 40 percent of a person's time.

34.5. Although adding people to a late software project can make it later, there are circumstances in which this is not true. Describe them.

34.6. The relationship between people and time is highly nonlinear. Using Putnam's software equation (described in Section 34.2.2), develop a table that relates number of people to project duration for a software project requiring 50,000 LOC and 15 person-years of effort (the productivity parameter is 5000 and $B = 0.37$). Assume that the software must be delivered in 24 months plus or minus 12 months.

34.7. Assume that you have been contracted by a university to develop an online course registration system (OLCRS). First, act as the customer (if you're a student, that should be easy) and specify the characteristics of a good system. (Alternatively, your instructor will provide you with a set of preliminary requirements for the system.) Using the estimation methods discussed in Chapter 33, develop an effort and duration estimate for OLCRS. Suggest how you would:

a. Define parallel work activities during the OLCRS project.
b. Distribute effort throughout the project.
c. Establish milestones for the project.

34.8. Select an appropriate task set for the OLCRS project.

34.9. Define a task network for OLCRS described in Problem 34.7, or alternatively, for another software project that interests you. Be sure to show tasks and milestones and to attach effort and duration estimates to each task. If possible, use an automated scheduling tool to perform this work.

34.10. If an automated scheduling tool is available, determine the critical path for the network defined in problem 34.9.

34.11. Using a scheduling tool (if available) or paper and pencil (if necessary), develop a time-line chart for the OLCRS project.

34.12. Assume you are a software project manager and that you've been asked to compute earned value statistics for a small software project. The project has 56 planned work tasks that are estimated to require 582 person-days to complete. At the time that you've been asked to do the earned value analysis, 12 tasks have been completed. However the project schedule indicates that 15 tasks should have been completed. The following scheduling data (in person-days) are available:

Task	Planned Effort	Actual Effort
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Compute the SPI, schedule variance, percent scheduled for completion, percent complete, CPI, and cost variance for the project.

Further Readings and Information Sources

Virtually every book written on software project management contains a discussion of scheduling. The Project Management Institute (*PMBOK Guide,* 5th ed., PMI, 2013), Wysoki (*Effective Project Management: Traditional, Agile, Extreme,* 6th ed., Wiley, 2011), Lewis (*Project Planning Scheduling and Control,* 5th ed., McGraw-Hill, 2010), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 10th ed., Wiley, 2009), Chemuturi and Cagley (*Mastering Software Project Management: Best Practices, Tools, and Techniques,* J. Ross Publishing, 2010), Hughes and Cotterel (*Software Project Management,* 5th ed., McGraw-Hill, 2009), Luckey and Phillips (*Software Project Management for Dummies,* For Dummies, 2006), Lewin (*Better Software Project Management,* Wiley, 2001), and Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3rd ed., Wiley, 2000) contain worthwhile discussions of the subject. Although application specific, Harris (*Planning and Scheduling Using Microsoft Office Project 2010,* Eastwood Harris Pty Ltd., 2010) provides a useful discussion of how scheduling tools can be used to successfully track and control a software project.

Fleming and Koppelman (*Earned Value Project Management,* 3rd ed., Project Management Institute Publications, 2010), Budd (*A Practical Guide to Earned Value Project Management*, 2nd ed., Management Concepts, 2009), and Webb and Wake (*Using Earned Value: A Project Manager's Guide,* Ashgate Publishing, 2003) discuss the use of earned value techniques for project planning, tracking, and control in considerable detail.

A wide variety of information sources on software project scheduling is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman.**

CHAPTER

35 Risk Management

Key Concepts

assessment. 781
identification. 780
projection 782
refinement 787
risk categories . . . 779
risk exposure 786
risk item checklist 781
risk table 783
RMMM. 790
safety and hazards 790
strategies. 778
  proactive 778
  reactive 778

In his book on risk analysis and management, Robert Charette [Cha89] presents a conceptual definition of risk:

> First, risk concerns future happenings. Today and yesterday are beyond active concern, as we are already reaping what was previously sowed by our past actions. The question is, can we, therefore, by changing our actions today, create an opportunity for a different and hopefully better situation for ourselves tomorrow. This means, second, that risk involves change, such as in changes of mind, opinion, actions, or places . . . [Third,] risk involves choice, and the uncertainty that choice itself entails. Thus paradoxically, risk, like death and taxes, is one of the few certainties of life.

When you consider risk in the context of software engineering, Charette's three conceptual underpinnings are always in evidence. The future is your concern—what risks might cause the software project to go awry? Change is your concern—how will changes in customer requirements, development

Quick Look

What is it? Risk analysis and management are a series of steps that help a software team understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Who does it? Everyone involved in the software process—managers, software engineers, and other stakeholders—participate in risk analysis and management.

Why is it important? Think about the Boy Scout motto: "Be prepared." Software is a difficult undertaking. Lots of things can go wrong, and frankly, many often do. It's for this reason that being prepared—understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software project management.

What are the steps? Recognizing what can go wrong is the first step, called "risk identification." Next, each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur. Once this information is established, risks are ranked, by probability and impact. Finally, a plan is developed to manage those risks that have high probability and high impact.

What is the work product? A risk mitigation, monitoring, and management (RMMM) plan or a set of risk information sheets is produced.

How do I ensure that I've done it right? The risks that are analyzed and managed should be derived from a thorough study of the people, the product, the process, and the project. The RMMM should be revisited as the project proceeds to ensure that risks are kept up to date. Contingency plans for risk management should be realistic.

technologies, target environments, and all other entities connected to the project affect timeliness and overall success? Last, you must grapple with choices—what methods and tools should you use, how many people should be involved, how much emphasis on quality is "enough"?

Peter Drucker [Dru75] once said, "While it is futile to try to eliminate risk, and questionable to try to minimize it, it is essential that the risks taken be the right risks." Before you can identify the "right risks" to be taken during a software project, it is important to identify all risks that are obvious to both managers and practitioners.

35.1 Reactive versus Proactive Risk Strategies

Quote:

"If you don't actively attack the risks, they will actively attack you."

Tom Gilb

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [Tho92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would react in some heroic way.

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire-fighting mode*. When this fails, "crisis management" [Cha92] takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A *proactive* strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management.

35.2 Software Risks

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics: *uncertainty*—the risk may or may not happen; that is, there are no

100 percent probable risks[1]—and *loss*—if the risk becomes a reality, unwanted consequences or losses will occur [Hig95]. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

? What types of risks are we likely to encounter as software is built?

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. In Chapter 33, project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

It is extremely important to note that simple risk categorization won't always work. Some risks are simply unpredictable in advance.

Quote:

"Projects with no real risks are losers. They are almost always devoid of benefit; that's why they weren't done years ago."

Tom DeMarco and Tim Lister

Another general categorization of risks has been proposed by Charette [Cha89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

1 A risk that is 100 percent probable is a constraint on the software project.

INFO

Seven Principles of Risk Management

The Software Engineering Institute (SEI) (**www.sei.cmu.edu**) identifies seven principles that "provide a framework to accomplish effective risk management." They are:

Maintain a global perspective—view software risks within the context of a system in which it is a component and the business problem that it is intended to solve.

Take a forward-looking view—think about the risks that may arise in the future (e.g., due to changes in the software); establish contingency plans so that future events are manageable.

Encourage open communication—if someone states a potential risk, don't discount it. If a risk is proposed in an informal manner, consider it. Encourage all stakeholders and users to suggest risks at any time.

Integrate—a consideration of risk must be integrated into the software process.

Emphasize a continuous process—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.

Develop a shared product vision—if all stakeholders share the same vision of the software, it is likely that better risk identification and assessment will occur.

Encourage teamwork—the talents, skills, and knowledge of all stakeholders should be pooled when risk management activities are conducted.

35.3 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

ADVICE

Although generic risks are important to consider, it's the product-specific risks that cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.

There are two distinct types of risks for each of the categories that have been presented in Section 35.2. *Generic risks* are a potential threat to every software project. *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. To identify product-specific risks, the project plan and the software statement of scope are examined, and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a *risk item checklist*. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- *Product size*—Risks associated with the overall size of the software to be built or modified.
- *Business impact*—Risks associated with constraints imposed by management or the marketplace.
- *Stakeholder characteristics*—Risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.

- *Process definition*—Risks associated with the degree to which the software process has been defined and is followed by the development organization.
- *Development environment*—Risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*—Risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*—Risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow you to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" [AFC88] are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

A number of comprehensive checklists for software project risk are available on the Web (e.g., [Baa07], [NAS07], [Wor04]). You can use these checklists to gain insight into generic risks for software projects. In addition to the use of checklists, *risk patterns* [Mil04] have been proposed as a systematic approach to risk identification.

35.3.1 Assessing Overall Project Risk

The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world [Kei98]. The questions are ordered by their relative importance to the success of a project.

? Is the software project we're working on at serious risk?

1. Have top software and customer managers formally committed to support the project?
2. Are end users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and its customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end users have realistic expectations?
6. Is the project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?

WebRef

Risk radar is a database and tools that help managers identify, rank, and communicate project risks. It can be found at **www.spmn.com.**

10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

35.3.2 Risk Components and Drivers

The U.S. Air Force [AFC88] has published a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components—performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

Quote:

"Risk management is project management for adults."

Tim Lister

- *Performance risk*—The degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *Cost risk*—The degree of uncertainty that the project budget will be maintained.
- *Support risk*—The degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *Schedule risk*—The degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Referring to Figure 35.1 [Boe89], a characterization of the potential consequences of errors (rows labeled 1) or a failure to achieve a desired outcome (rows labeled 2) are described. The impact category is chosen based on the characterization that best fits the description in the table.

35.4 Risk Projection

Risk projection, also called *risk estimation,* attempts to rate each risk in two ways—(1) the likelihood or probability that the risk is real and will occur and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the perceived likelihood of a risk.
2. Delineate the consequences of the risk.

FIGURE 35.1

Impact assessment

Source: [Boe89].

Category \ Components		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: (1) The potential consequence of undetected software errors or faults.
(2) The potential consequence if the desired outcome is not achieved.

3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.

35.4.1 Developing a Risk Table

A risk table provides you with a simple technique for risk projection.[2] A sample risk table is illustrated in Figure 35.2.

2 The risk table can be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries.

FIGURE 35.2

Sample risk table prior to sorting

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

ADVICE

Think hard about the software you're about to build and ask yourself, "what can go wrong?" Create your own list and ask other members of the team to do the same.

You begin by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists referenced in Section 35.3. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge.

Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented in Figure 35.1, and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged[3] to determine an overall impact value.

KEY POINT

A risk table is sorted by probability and impact to rank risks.

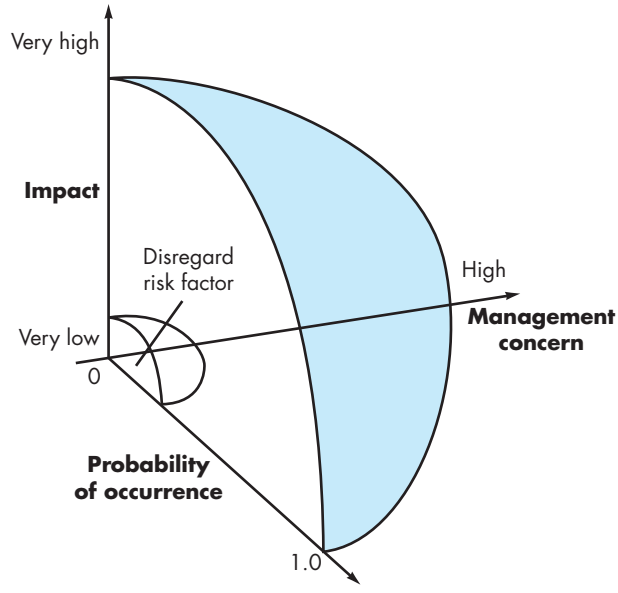
Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

You can study the resultant sorted table and define a cutoff line. The *cutoff line* (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are

3 A weighted average can be used if one risk component has more significance for a project.

FIGURE 35.3

Risk and management concern



reevaluated to accomplish second-order prioritization. Referring to Figure 35.3, risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cutoff line should be managed. The column labeled RMMM contains a pointer into a *risk mitigation, monitoring, and management plan* or, alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 35.5 and 35.6.

Quote:

"[Today,] no one has the luxury of getting to know a task so well that it holds no surprises, and surprises mean risk."

Stephen Grey

Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed (e.g., [McC09]).

35.4.2 Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how

much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, you want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.

How do we assess the consequences of a risk?

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], you can apply the following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) using Figure 35.1, determine the impact for each component based on the criteria shown, and (3) complete the risk table and analyze the results as described in the preceding sections.

The overall *risk exposure,* RE, is determined using the following relationship [Hal98]:

$$RE = P \times C$$

where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. Eighty percent (likely).

Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 × 100 × 14 = $25,200.

Risk exposure. RE = 0.80 × 25,200 ~ $20,200.

ADVICE

Compare RE for all risks to the cost estimate for the project. If RE is greater than 50 percent of the project cost, the viability of the project must be evaluated.

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques described in Sections 35.4.1 and 35.4.2 are applied iteratively as the software project proceeds.[4] The project team should revisit the risk table at regular intervals, reevaluating each risk to

4 If you have further interest, a more mathematical treatment of the cost of risk is presented in [Ben10].

determine when new circumstances cause its probability and impact to change. As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.

SafeHome

Risk Analysis

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I'd like to spend some time brainstorming risks for the *SafeHome* project.

Jamie: As in what can go wrong?

Doug: Yep. Here are a few categories where things can go wrong.

[He shows everyone the categories noted in the introduction to Section 35.3.]

Vinod: Umm . . . do you want us to just call them out, or . . .

Doug: No here's what I thought we'd do. Everyone make a list of risks . . . right now . . .

[Ten minutes pass, everyone is writing.]

Doug: Okay, stop.

Jamie: But I'm not done!

Doug: That's okay. We'll revisit the list again. Now, for each item on your list, assign a percent likelihood that the risk will occur. Then, assign an impact to the project on a scale of 1 (minor) to 5 (catastrophic).

Vinod: So if I think that the risk is a coin flip, I specify a 50 percent likelihood, and if I think it'll have a moderate project impact, I specify a 3, right?

Doug: Exactly.

[Five minutes pass, everyone is writing.]

Doug: Okay, stop. Now we'll make a group list on the whiteboard. I'll do the writing; we'll call out one entry from your list in round-robin format.

[Fifteen minutes pass; the list is created.]

Jamie (pointing at the board and laughing): Vinod, that risk (pointing toward an entry on the board) is ridiculous. There's a higher likelihood that we'll all get hit by lightning. We should remove it.

Doug: No, let's leave it for now. We consider all risks, no matter how weird. Later we'll winnow the list.

Jamie: But we already have over 40 risks . . . how on earth can we manage them all?

Doug: We can't. That's why we'll define a cut-off after we sort these guys. I'll do that off-line and we'll meet again tomorrow. For now, get back to work . . . and in your spare time, think about any risks that we've missed.

35.5 Risk Refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

? What's a good way to describe a risk?

One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format [Glu94]. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 35.4.2, you could write:

> Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

35.6 Risk Mitigation, Monitoring, and Management

Quote:

"If I take so many precautions, it is because I leave nothing to chance."

Napolean

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation*. For example, assume that high staff turnover is noted as a project risk r_1. Based on past history and management intuition, the likelihood l_1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x_1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:

? What can we do to mitigate a risk?

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.

- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, *risk-monitoring* activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents or Wikis," and/or meeting with other team members who will remain on the project.

ADVICE

If RE for a specific risk is less than the cost of risk mitigation, don't try to mitigate the risk but continue to monitor it.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, you perform a classic cost-benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself. For this reason, you should adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help you to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

Software safety and hazard analysis (e.g., [Dun02], [Her00], [Lev95]) are software quality assurance activities (Chapter 21) that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

35.7 The RMMM Plan

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate *risk mitigation, monitoring and management plan*. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS) [Wil97]. In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. The format of the RIS is illustrated in Figure 35.4.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin [what risk(s) caused which problems throughout the project].

FIGURE 35.4

Risk information sheet
Source: [Wil97].

Risk information sheet			
Risk ID: P02-4-32	Date: 5/9/09	Prob: 80%	Impact: high
Description: Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
Refinement/context: Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
Mitigation/monitoring: 1. Contact third party to determine conformance with design standards. 2. Press for interface standards completion; consider component structure when deciding on interface protocol. 3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.			
Management/contingency plan/trigger: *RE* computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7/1/09.			
Current status: 5/12/09: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	

SOFTWARE TOOLS

Risk Management

Objective: The objective of risk management tools is to assist a project team in defining risks, assessing their impact and probability, and tracking risks throughout a software project.

Mechanics: In general, risk management tools assist in generic risk identification by providing a list of typical project and business risks, provide checklists or other "interview" techniques that assist in identifying project specific risks, assign probability and impact to each risk, support risk mitigation strategies, and generate many different risk-related reports.

Representative Tools:[5]

@risk, developed by Palisade Corporation (**www.palisade.com**), is a generic risk analysis tool that uses Monte Carlo simulation to drive its analytical engine.

Riskman, distributed by ABS Consulting (**www.absconsulting.com/riskmansoftware/index.html**), is a risk evaluation expert system that identifies project-related risks.

5 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

Risk Radar, developed by SPMN (**www.spmn.com**), assists project managers in identifying and managing project risks.

ARM, developed by Deltek (**www.deltek.com**), Web-based tool that allows suppliers, customers and geographically disbursed project teams to collaborate on essential risk knowledge.

X:PRIMER, developed by GrafP Technologies (**www.grafp.com**) is a generic Web-based tool that predicts what can go wrong on a project and identifies root causes for potential failures and effective countermeasures.

35.8 Summary

Whenever a lot is riding on a software project, common sense dictates risk analysis. And yet, most software project managers do it informally and superficially, if they do it at all. The time spent identifying, analyzing, and managing risk pays itself back in many ways—less upheaval during the project, a greater ability to track and control a project, and the confidence that comes with planning for problems before they occur.

Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it. To quote Sun Tzu, a Chinese general who lived 2,500 years ago, "If you know the enemy and know yourself, you need not fear the result of a hundred battles." For the software project manager, the enemy is risk.

Problems and Points to Ponder

35.1. Provide five examples from other fields that illustrate the problems associated with a reactive risk strategy.

35.2. Describe the difference between "known risks" and "predictable risks."

35.3. Add three additional questions or topics to each of the risk item checklists presented at the SEPA website.

35.4. You've been asked to build software to support a low-cost video editing system. The system accepts digital video as input, stores the video on disk, and then allows the user to do a wide range of edits to the digitized video. The result can then be output to DVD or other media. Do a small amount of research on systems of this type and then make a list of technology risks that you would face as you begin a project of this type.

35.5. You're the project manager for a major software company. You've been asked to lead a team that's developing "next generation" word-processing software. Create a risk table for the project.

35.6. Describe the difference between risk components and risk drivers.

35.7. Develop a risk mitigation strategy and specific risk mitigation activities for three of the risks noted in Figure 35.2.

35.8. Develop a risk monitoring strategy and specific risk monitoring activities for three of the risks noted in Figure 35.2. Be sure to identify the factors that you'll be monitoring to determine whether the risk is becoming more or less likely.

35.9. Develop a risk management strategy and specific risk management activities for three of the risks noted in Figure 35.2.

35.10. Attempt to refine three of the risks noted in Figure 35.2, and then create risk information sheets for each.

35.11. Represent three of the risks noted in Figure 35.2 using a CTC format.

35.12. Recompute the risk exposure discussed in Section 35.4.2 when cost/LOC is $16 and the probability is 60 percent.

35.13. Can you think of a situation in which a high-probability, high-impact risk would not be considered as part of your RMMM plan?

35.14. Describe five software application areas in which software safety and hazard analysis would be a major concern.

Further Readings and Information Sources

The software risk management literature has expanded significantly over the past few decades. Mun (*Modeling Risk,* 2nd ed., Wiley, 2010) presents a detailed mathematical treatment of risk analysis that can be applied to software projects. Mulcahy (*Risk Management, Tricks of the Trade for Project Managers,* 2nd ed., RMC Publications, 2010), Kendrick (*Identifying and Managing Project Risk,* 2nd ed., American Management Association, 2009), Crohy and his colleagues (*The Essentials of Risk Management,* McGraw-Hill, 2006), and Marrison (*The Fundamentals of Risk Measurement,* McGraw-Hill, 2002) present useful methods and tools that every project manager can use. Jindal and his colleagues (*Risk Management in Software Engineering,* Create Space Independent Publishing, 2012) discuss embedding security risk assessment as part system development.

DeMarco and Lister (*Dancing with Bears,* Dorset House, 2003) have written an entertaining and insightful book that guides software managers and practitioners through risk management. Moynihan (*Coping with IT/IS Risk Management,* Springer-Verlag, 2002) presents pragmatic advice from project managers who deal with risk on a continuing basis. Royer (*Project Risk Management,* Management Concepts, 2002) and Smith and Merritt (*Proactive Risk Management,* Productivity Press, 2002) suggest a proactive process for risk management. Karolak (*Software Engineering Risk Management,* Wiley, 2002) has written a guidebook that introduces an easy-to-use risk analysis model with worthwhile checklists and questionnaires supported by a software package.

Capers Jones (*Assessment and Control of Software Risks,* Prentice Hall, 1994) presents a detailed discussion of software risks that includes data collected from hundreds of software projects. Jones defines 60 risk factors that can affect the outcome of software projects. Boehm [Boe89] suggests excellent questionnaire and checklist formats that can prove invaluable in identifying risk. Charette [Cha89] presents a detailed treatment of the mechanics of risk analysis, calling on probability theory and statistical techniques to analyze risks. In a companion volume, Charette (*Application Strategies for Risk Analysis,* McGraw-Hill, 1990) discusses risk in the context of both system and software engineering and suggests pragmatic strategies for risk management. Gilb (*Principles of Software Engineering Management,* Addison-Wesley, 1988) presents a set of "principles" (which are often amusing and sometimes profound) that can serve as a worthwhile guide for risk management.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects,* MIT Press, 2003) and Yourdon (*Death March,* Prentice Hall, 1997) discuss what happens when

risks overwhelm a software project team. Bernstein (*Against the Gods,* Wiley, 1998) presents an entertaining history of risk that goes back to ancient times.

The Software Engineering Institute has published many detailed reports and guidebooks on risk analysis and management. The Air Force Systems Command pamphlet AFSCP 800-45 [AFC88] describes risk identification and reduction techniques. Every issue of the *ACM Software Engineering Notes* has a section entitled "Risks to the Public" (editor, P. G. Neumann). If you want the latest and best software horror stories, this is the place to go.

A wide variety of information sources on software risk management is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman.**

CHAPTER

36

Maintenance and Reengineering

Key Concepts

business process reengineering (BPR) 799
document restructuring 804
forward engineering 811
inventory analysis . 803
maintainability . . . 797

Regardless of its application domain, its size, or its complexity, computer software will evolve over time. Change drives this process. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context. Over the past 40 years, Manny Lehman [e.g., Leh97a] and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution*. The

Quick Look

What is it? Consider any technology product that has served you well. You use it regularly, but it's getting old. It breaks too often, takes longer to repair than you'd like, and no longer represents the newest technology. What to do? For a time, you try to fix it, patch it, even extend its functionality. That's called maintenance. But maintenance becomes increasingly difficult as the years pass. There comes a time when you'll need to rebuild it. You'll create a product with added functionality, better performance and reliability, and improved maintainability. That's what we call reengineering.

Who does it? At an organizational level, maintenance is performed by support staff that are part of the software engineering organization. Reengineering is performed by business specialists (often consulting companies), and at the software level, reengineering is performed by software engineers.

Why is it important? We live in a rapidly changing world. The demands on business functions and the information technology that supports them are changing at a pace that puts enormous competitive pressure on every commercial organization. That's why software must be maintained continually, and at the appropriate time, reengineered to keep pace.

What are the steps? Maintenance corrects defects, adapts the software to meet a changing environment, and enhances functionality to meet the evolving needs of customers. At a strategic level, business process reengineering (BPR) defines business goals, identifies and evaluates existing business processes, and creates revised business processes that better meet current goals. Software reengineering encompasses inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability.

What is the work product? A variety of maintenance and reengineering work products (e.g., use cases, analysis and design models, test procedures) are produced. The final output is upgraded software.

How do I ensure that I've done it right? Use the same SQA practices that are applied in every software engineering process—technical reviews assess the analysis and design models; specialized reviews consider business applicability and compatibility; and testing is applied to uncover errors in content, functionality, and interoperability.

restructuring. 809
code 809
data 810
reverse engineering. 805
data 807
processing 807
user interfaces . 808
software maintenance 796
software reengineering 802
supportability 798

details of this work are beyond the scope of this book, but the underlying laws that have been derived are worthy of note [Leh97b]:

The Law of Continuing Change (1974): Software that has been implemented in a real-world computing context and will therefore evolve over time (called *E-type systems*) must be continually adapted else they become progressively less satisfactory.

The Law of Increasing Complexity (1974): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

The Law of Self Regulation (1974): The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

The Law of Conservation of Organizational Stability (1980): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

? Why do legacy systems evolve as time passes?

The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The Law of Continuing Growth (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The Feedback System Law (1996): E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality. In this chapter, we'll discuss the challenge of software maintenance and the reengineering activities that are required to extend the effective life of legacy systems.

36.1 Software Maintenance

It begins almost immediately. Software is released to end users, and within days, bug reports filter back to the software engineering organization. Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment. And within months, another corporate group that wanted nothing to do with the software when it was released now recognizes that it may provide unexpected benefit. They'll need a few enhancements to make it work in their world.

The challenge of software maintenance has begun. You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has

grown long, and the work it implies threatens to overwhelm the available resources. As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications. In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance.

You may ask why so much maintenance is required and why so much effort is expended. Osborne and Chikofsky [Osb90] provide a partial answer:

> Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture. The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running . . .

Another reason for the software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, other generations of software people have modified the system and moved on. And today, there may be no one left who has any direct knowledge of the legacy system.

As we noted in Chapter 29, the ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, you must develop mechanisms for evaluating, controlling, and making modifications.

Quote:

"Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain.

Gerald Berns

Throughout this book, we've emphasized the importance of understanding the problem (analysis) and developing a well-structured solution (design). In fact, Part 2 of the book is dedicated to the mechanics of these software engineering actions, and Part 3 focuses on the techniques required to be sure you've done them correctly. Both analysis and design lead to an important software characteristic that we call maintainability. In essence, *maintainability* is a qualitative indication[1] of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability.

But what is maintainability? Maintainable software exhibits effective modularity (Chapter 12). It makes use of design patterns (Chapter 16) that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. It has undergone a variety of quality assurance techniques (Part 3 of this book) that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they

1 There are many quantitative measures that provide an indirect indication of maintainability (e.g., [Sch99], [SEI02]).

may not be around when changes must be made. Therefore, the design and implementation of the software must "assist" the person who is making the change.

36.2 Software Supportability

In order to effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life cycle of the software. A reasonable definition of *software supportability* is

> . . . the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any other resource required to maintain the software operational and capable of satisfying its function. [SSO08]

In essence, supportability is one of many quality factors that should be considered during the analysis and design actions that are part of the software process. It should be addressed as part of the requirements model (or specification) and considered as the design evolves and construction commences.

WebRef

A wide array of downloadable documents on software supportability can be found at **www.software-supportability.org/Downloads.html**.

For example, the need to "antibug" software at the component and code level has been discussed previously in the book. The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects *will* be encountered). In addition, support personnel should have access to a database that contains records of all defects that have already been encountered—their characteristics, cause, and cure. This will enable support personnel to examine "similar" defects and may provide a means for more rapid diagnosis and correction.

Although defects encountered in an application are a critical support issue, supportability also demands that resources be provided to support day-to-day end-user issues. The job of end-user support personnel is to answer user queries about the installation, operation, and use of the application.

36.3 Reengineering

In a seminal article written for the *Harvard Business Review,* Michael Hammer [Ham90] laid the foundation for a revolution in management thinking about business processes and computing:

> It is time to stop paving the cow paths. Instead of embedding outdated processes in silicon and software, we should obliterate them and start over. We should "reengineer" our businesses: use the power of modern information technology to radically

redesign our business processes in order to achieve dramatic improvements in their performance.

Quote:

"To face tomorrow with the thought of using the methods of yesterday is to envision life at a standstill."

James Bell

The hype associated with reengineering waned, but the process itself continues in companies large and small. The nexus between business reengineering and software engineering lies in a "system view."

As managers work to modify business rules to achieve greater effectiveness and competitiveness, software must keep pace. In some cases, this means the creation of major new computer-based systems.[2] But in many others, it means the modification or rebuilding of existing applications.

In the sections that follow, we examine reengineering in a top-down manner, beginning with a brief overview of business process reengineering and proceeding to a more detailed discussion of the technical activities that occur when software is reengineered.

36.4 Business Process Reengineering

KEY POINT

BPR often results in new software functionality, whereas software reengineering works to replace existing software functionality with better, more maintainable software.

Business process reengineering (BPR) extends far beyond the scope of information technologies and software engineering. Among the many definitions (most somewhat abstract) that have been suggested for BPR is one published in *Fortune* magazine [Ste93]: "[T]he search for, and the implementation of, radical change in business process to achieve breakthrough results." But how is the search conducted, and how is the implementation achieved? More important, how can we ensure that the "radical change" suggested will in fact lead to "breakthrough results" instead of organizational chaos?

36.4.1 Business Processes

ADVICE

As a software engineer, your work occurs at the bottom of this hierarchy. Be sure, however, that someone has given serious thought to the level above. If this hasn't been done, your work is at risk.

A business process is "a set of logically related tasks performed to achieve a defined business outcome" [Dav90]. Within the business process, people, equipment, material resources, and business procedures are combined to produce a specified result. Examples of business processes include designing a new product, purchasing services and supplies, hiring a new employee, and paying suppliers. Each demands a set of tasks, and each draws on diverse resources within the business.

Every business process has a defined customer—a person or group that receives the outcome (e.g., an idea, a report, a design, a service, a product). In addition, business processes cross organizational boundaries. They require that different organizational groups participate in the "logically related tasks" that define the process.

2 The explosion of Web and mobile applications and systems is indicative of this trend.

Every system is actually a hierarchy of subsystems. A business is no exception. The overall business is segmented in the following manner:

The business → business systems → business processes → business subprocesses

Each business system (also called *business function*) is composed of one or more business processes, and each business process is defined by a set of subprocesses.

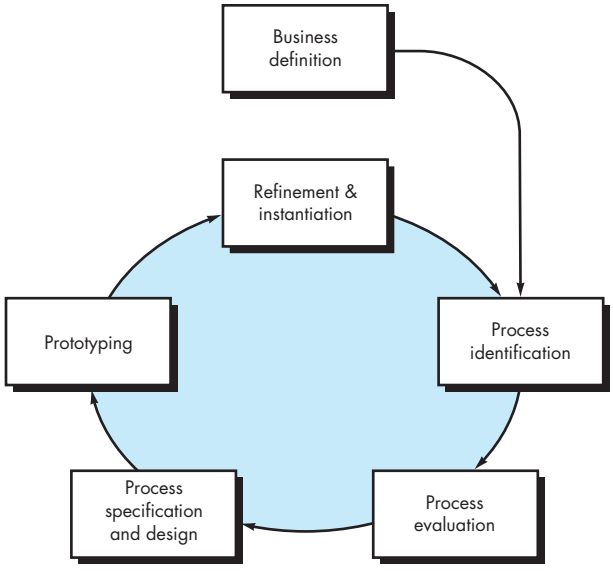
BPR can be applied at any level of the hierarchy, but as the scope of BPR broadens (i.e., as we move upward in the hierarchy), the risks associated with BPR grow dramatically. For this reason, most BPR efforts focus on individual processes or subprocesses.

36.4.2 A BPR Model

Like most engineering activities, business process reengineering is iterative. Business goals and the processes that achieve them must be adapted to a changing business environment. For this reason, there is no start and end to BPR—it is an evolutionary process. A model for business process reengineering is depicted in Figure 36.1. The model defines six activities:

1. **Business definition.** Business goals are identified within the context of four key drivers: cost reduction, time reduction, quality improvement, and personnel development and empowerment. Goals may be defined at the business level or for a specific component of the business.
2. **Process identification.** Processes that are critical to achieving the goals defined in the business definition are identified. They may then be ranked

FIGURE 36.1
A BPR model



by importance, by need for change, or in any other way that is appropriate for the reengineering activity.

Quote:

"As soon as we are shown something old in a new thing, we are pacified."

F. W. Nietzsche

3. **Process evaluation.** The existing process is thoroughly analyzed and measured. Process tasks are identified; the costs and time consumed by process tasks are noted; and quality/performance problems are isolated.
4. **Process specification and design.** Based on information obtained during the first three BPR activities, use cases (Chapters 8 and 9) are prepared for each process that is to be redesigned. Within the context of BPR, use cases identify a scenario that delivers some outcome to a customer. With the use case as the specification of the process, a new set of tasks are designed for the process.
5. **Prototyping.** A redesigned business process must be prototyped before it is fully integrated into the business. This activity "tests" the process so that refinements can be made.
6. **Refinement and instantiation.** Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

These BPR activities are sometimes used in conjunction with workflow analysis tools. The intent of these tools is to build a model of existing workflow in an effort to better analyze existing processes.

SOFTWARE TOOLS

Business Process Reengineering (BPR)

Objective: The objective of BPR tools is to support the analysis and assessment of existing business processes and the specification and design of new ones.

Mechanics: Tools mechanics vary. In general, BPR tools allow a business analyst to model existing business processes in an effort to assess workflow inefficiencies or functional problems. Once existing problems are identified, tools allow the analysis to prototype and/or simulate revised business processes.

Representative Tools:[3]

ExtendSim, developed by ImagineThat (**www.imaginethatinc.com**), is a simulation tool for modeling existing processes and exploring new ones. Extend provides comprehensive what-if capability that enables a business analysis to explore different process scenarios.

Metastrom BPM, developed by OpentText (**http://bps.opentext.com/**), provides business process management support for both manual and automated processes.

IceTools, developed by Blue Ice (**http://www.icetools.com/home.html**), is a collection of BPR templates for Microsoft Office and Microsoft Visio.

OMNIBUS, developed by Kovair (**http://www.kovair.com/**), is one of many tools that enable an organization to model process workflow (in this case, IT workflow).

ProcessMaker, open-source workflow suite developed by Colosa (**http://www.processmaker.com/**), incorporates a suite of tools for workflow modeling, simulation, and scheduling.

A useful list of BPR tool links can be found at **www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents**.

3 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

36.5 Software Reengineering

The scenario is all too common: An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the urgency of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software reengineering has been spawned by software maintenance problems that have been building for almost half a century.

36.5.1 A Software Reengineering Process Model

Reengineering takes time, it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering.

WebRef

An excellent source of information on software reengineering can be found at **reengineer.org**.

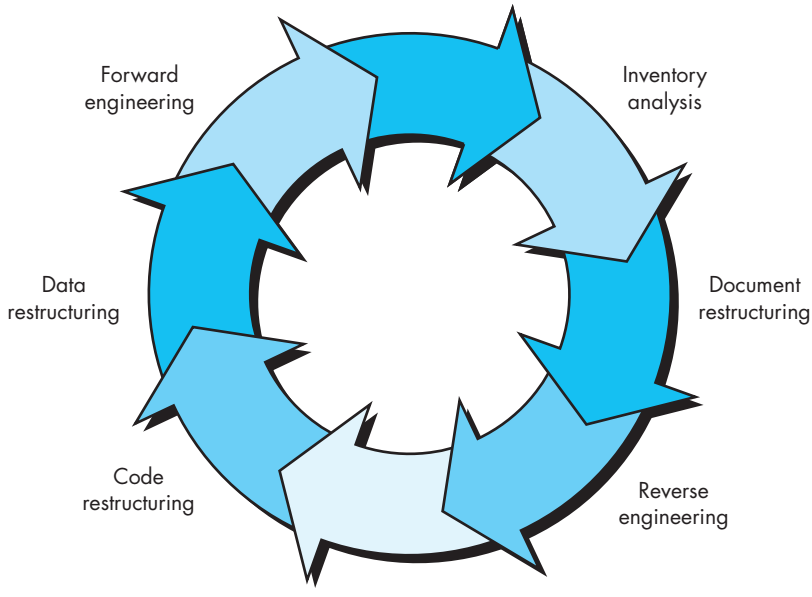
A workable strategy is encompassed in a reengineering process model. We'll discuss the model later in this section, but first, some basic principles.

Reengineering is a rebuilding activity. To better understand it, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).
- Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.

FIGURE 36.2

A software reengineering process model



- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

To implement these principles, you can use a software reengineering process model that defines six activities, shown in Figure 36.2. In some cases, these activities occur in a linear sequence, but this is not always the case. For example, it may be that reverse engineering (understanding the internal workings of a program) may have to occur before document restructuring can commence.

36.5.2 Software Reengineering Activities

ADVICE

If time and resources are in short supply, you might consider applying the Pareto principle to the software that is to be reengineered. Apply the reengineering process to the 20 percent of the software that accounts for 80 percent of the problems.

The reengineering paradigm shown in Figure 36.2 is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular basis. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

Document restructuring. Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options? In some cases, creating documentation when none exists is simply too costly. If the software works, let it be! In other cases, some documentation must be created, but only when changes are made. If a modification occurs, document it. Finally, there are situations in which a critical system must be fully documented, but even here, documents should achieve an essential minimum. Your software organization must choose the documentation option that is most appropriate for each case.

ADVICE

Create only as much documentation as you need to understand the software, not one page more.

Reverse engineering. Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of *design recovery*. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring. The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is *code restructuring*.[4] Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically) or even rewritten in a more modern programming language. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring. A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

4 Code restructuring has some of the elements of "refactoring," a redesign concept introduced in Chapter 12 and discussed elsewhere in this book.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering. In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software recreates the function of the existing system and also adds new functions and/or improves overall performance.

36.6 Reverse Engineering

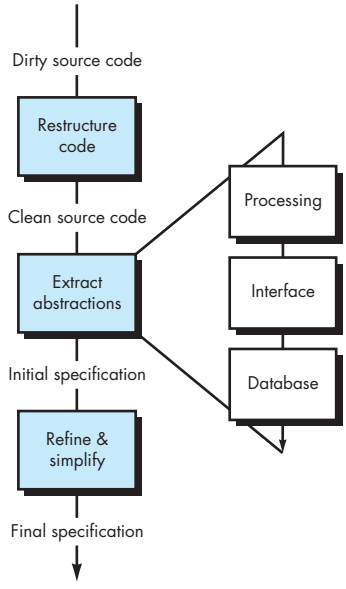
Reverse engineering conjures an image of the "magic slot." You feed a haphazardly designed, undocumented source file into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of abstraction), object models, data and/or control flow models (a relatively high level of abstraction), and data models (a high level of abstraction). As the abstraction level increases, you are provided with information that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

FIGURE 36.3

The reverse engineering process



KEY POINT

Three reverse engineering issues must be addressed: abstraction level, completeness, and directionality.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

WebRef

Useful resources for "design recovery and program understanding" can be found at **http://www.softpanorama.net/SE/reverse_engineering_links.shtml.**

The reverse engineering process is represented in Figure 36.3. Before reverse engineering activities can commence, unstructured ("dirty") source code is restructured (Section 36.5.1) so that it contains only the structured programming constructs.[5] This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

The core of reverse engineering is an activity called *extract abstractions*. You must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

5 Code can be restructured using a *restructuring engine*—a tool that restructures source code.

36.6.1 Reverse Engineering to Understand Data

ADVICE

In some cases, the first reengineering activity attempts to construct a UML class diagram.

Reverse engineering of data occurs at different levels of abstraction and is often the first reengineering task. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new systemwide database.

ADVICE

The approach to reverse engineering for data for conventional software follows an analogous path: (1) build a data model, (2) identify attributes of data objects, and (3) define relationships.

Internal data structures. Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables. In many cases, the data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes.

Database structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [Pre94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys), (3) refine the tentative classes, (4) define generalizations, and (5) discover associations using techniques that are analogous to the CRC approach. Once information defined in the preceding steps is known, a series of transformations [Pre94] can be applied to map the old database structure into a new database structure.

Quote:

"There exists a passion for comprehension, just as there exists a passion for music. That passion is rather common in children, but gets lost in most people later on."

Albert Einstein

36.6.2 Reverse Engineering to Understand Processing

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within a larger system. Each of the programs that make up the system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each

component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system, program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[6]

Things become more complex when the code inside a component is considered. You should look for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, you can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools can be used to help you understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

36.6.3 Reverse Engineering User Interfaces

Sophisticated GUIs have become de rigueur for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [Mer93] suggest three basic questions that must be answered as reverse engineering of the UI commences:

? How do I understand the working of an existing user interface?

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a "replacement," or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation (Chapter 11) can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioral model must be extracted from the code.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop a new interaction metaphor. For example, an old UI requests that a user

6 Often, specifications written early in the life history of a program are never updated. As changes are made, the code no longer conforms to the specification.

provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a touch-screen slide bar to accomplish the same function.

SOFTWARE TOOLS

Reverse Engineering

Objective: To help software engineers understand the internal design structure of complex programs.

Mechanics: In most cases, reverse engineering tools accept source code as input and produce a variety of structural, procedural, data, and behavioral design representations.

Representative Tools:[7]

Imagix 4D, developed by Imagix (**www.imagix.com**), "helps software developers understand complex or legacy C and C++ software" by reverse engineering and documenting source code.

Understand, developed by Scientific Toolworks (**www.scitools.com**), parse Ada, Fortran, C, C++, C#, PHP, HTML, JavaScript, Python, and Java "to reverse-engineer, automatically document, calculate code metrics, and help you understand, navigate and maintain source code."

A list of reverse engineering tools can be found at **http://www.eclipse.org/gmt/modisco/relatedProjects.php**.

36.7 RESTRUCTURING

Software restructuring modifies source code and/or data in an effort to make it amenable to future changes. In general, restructuring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the restructuring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering (Section 36.8).

Restructuring occurs when the basic architecture of an application is solid, even though technical internals need work. It is initiated when major parts of the software are serviceable and only a subset of all modules and data need extensive modification.[8]

ADVICE

Although code restructuring can alleviate immediate problems associated with debugging or small changes, it is not reengineering. Real benefit is achieved only when data and architecture are restructured.

36.7.1 Code Restructuring

Code restructuring is performed to yield a design that produces the same function but with higher quality than the original program. In general, code restructuring techniques (e.g., Warnier's logical simplification techniques [War74]) model program logic using Boolean algebra and then apply a series of transformation rules that yield restructured logic. The objective is to take "spaghetti-bowl" code and derive a procedural design that conforms to the structured programming philosophy (Chapter 19).

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

8 It is sometimes difficult to make a distinction between extensive restructuring and redevelopment. Both are reengineering.

Other restructuring techniques have also been proposed for use with reengineering tools. A resource exchange diagram maps each program module and the resources (data types, procedures, and variables) that are exchanged between it and other modules. By creating representations of resource flow, the program architecture can be restructured to achieve minimum coupling among modules.

36.7.2 Data Restructuring

Before data restructuring can begin, a reverse engineering activity called *analysis of source code* should be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis*.

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When restructuring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

SOFTWARE TOOLS

Software Restructuring

Objective: The objective of restructuring tools is to transform older unstructured computer software into modern programming languages and design structures.

Mechanics: In general, source code is input and transformed into a better structured program. In some cases, the transformation occurs within the same programming language. In other cases, an older programming language is transformed into a more modern language.

Representative Tools[9]

DMS Software Reengineering Toolkit, developed by Semantic Design (**www.semdesigns.com**), provide a variety of restructuring capabilities for COBOL, C/C++, Java, Fortran 90, and VHDL.

Clone Doctor, developed by Semantic Designs (**www.semdesigns.com**), analyzes and transforms programs written in C, C++, Java, or COBOL or any other text-based computer language.

plusFORT, developed by Polyhedron (**www.polyhedron.com**), is a suite of FORTRAN tools that contains capabilities for restructuring poorly designed FORTRAN programs into the modern FORTRAN or C standard.

Pointers to a variety of reengineering and reverse engineering tools can be found at **http://www.comp.lancs.ac.uk/projects/renaissance/RenaissanceWeb/Reengineering/Tools.html** and **http://www.fujaba.de/projects.html**.

9 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

36.8 Forward Engineering

A program with control flow that is the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements. You have the following options:

? What options exist when we're faced with a poorly designed and implemented program?

1. You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.
2. You can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. You can redesign, recode, and test those portions of the software that require modification, applying a software engineering approach to all revised segments.
4. You can completely redesign, recode, and test the program, using reengineering tools to assist us in understanding the current design.

There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied.

ADVICE

Reengineering is a lot like getting your teeth cleaned. You can think of a thousand reasons to delay it, and you'll get away with procrastinating for quite a while. But eventually, your delaying tactics will come back to cause pain.

At first glance, the suggestion that you redevelop a large program when a working version already exists may seem quite extravagant. Before passing judgment, consider the following arguments. The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line. In addition, redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance. Because a prototype of the software already exists, development productivity should be much higher than average. The user now has experience with the software. Therefore, new requirements and the direction of change can be ascertained with greater ease. Automated tools for reengineering will facilitate some parts of the job. And finally, a complete software configuration (documents, programs, and data) will exist upon completion of preventive maintenance.

A large in-house software developer (e.g., a business systems software development group for a large consumer products company) may have 500 to 2,000 production programs within its domain of responsibility. These programs can be ranked by importance and then reviewed as candidates for forward engineering.

In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

36.8.1 Forward Engineering for Client-Server Architectures

Over the past few decades, centralized computing resources (including software) have been distributed among many client platforms. Although a variety of different distributed environments can be designed, the typical centralized application that has been reengineered into a client-server architecture has the following features: application functionality migrates to each client computer, new GUI interfaces are implemented at the client sites, database functions are allocated to the server, specialized functionality (e.g., compute-intensive analysis) may remain at the server site, and new communications, security, archiving, and control requirements must be established at both the client and server sites. It is important to note that the migration from centralized computing to client-server computing requires both business and software reengineering.

ADVICE

In some cases, migration to a client-server architecture should be approached not as reengineering, but as a new development effort. Reengineering enters the picture only when specific functionality from the old system is to be integrated into the new architecture.

Reengineering for client-server applications begins with a thorough analysis of the business environment that encompasses the existing mainframe. Three layers of abstraction can be identified. The *database sits* at the foundation of a client-server architecture and manages transactions and queries from server applications. Yet these transactions and queries must be controlled within the context of a set of business rules (defined by an existing or reengineered business process). Client applications provide targeted functionality to the user community.

The functions of the existing database management system and the data architecture of the existing database must be reverse engineered as a precursor to the redesign of the database foundation layer. The client-server database is reengineered to ensure that transactions are executed in a consistent manner, that all updates are performed only by authorized users, that core business rules are enforced (e.g., before a vendor record is deleted, the server ensures that no related accounts payable, contracts, or communications exist for that vendor), that queries can be accommodated efficiently, and that full archiving capability has been established.

The business rules layer represents software resident at both the client and the server. This software performs control and coordination tasks to ensure that transactions and queries between the client application and the database conform to the established business process.

The client applications layer implements business functions that are required by specific groups of end users. In many instances, an older centralized application is segmented into a number of smaller, reengineered desktop applications.

Communication among the desktop applications (when necessary) is controlled by the business rules layer.

A comprehensive discussion of client-server software design and reengineering is best left to books dedicated to the subject. If you have further interest, see [Van02], [Cou00], or [Orf99].

36.8.2 Forward Engineering for Object-Oriented Architectures

Object-oriented software engineering has become the development paradigm of choice for many software organizations. But what about existing applications that were developed using conventional methods? In some cases, the answer is to leave such applications "as is." In others, older applications must be reengineered so that they can be easily integrated into large, object-oriented systems.

Reengineering conventional software into an object-oriented implementation uses many of the same techniques discussed in Part 2 of this book. First, the existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created. If the reengineered system extends the functionality or behavior of the original application, use cases (Chapters 8 and 9) are created. The data models created during reverse engineering are then used in conjunction with CRC modeling (Chapter 10) to establish the basis for the definition of classes. Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.

As object-oriented forward engineering progresses from analysis to design, a CBSE process model (Chapter 10) can be invoked. If the existing application resides within a domain that is already populated by many object-oriented applications, it is likely that a robust component library exists and can be used during forward engineering.

For those classes that must be engineered from scratch, it may be possible to reuse algorithms and data structures from the existing conventional application. However, these must be redesigned to conform to the object-oriented architecture.

Quote:

"You can pay a little now, or you can pay a lot more later."

Sign in an auto dealership suggesting a tune-up

36.9 The Economics of Reengineering

In a perfect world, every unmaintainable program would be retired immediately, to be replaced by high-quality, reengineered applications developed using modern software engineering practices. But we live in a world of limited resources. Reengineering drains resources that can be used for other business purposes. Therefore, before an organization attempts to reengineer an existing application, it should perform a cost-benefit analysis.

A cost-benefit analysis model for reengineering has been proposed by Sneed [Sne95]. Nine parameters are defined:

P_1 = Current annual maintenance cost for an application
P_2 = Current annual operations cost for an application
P_3 = Current annual business value of an application
P_4 = Predicted annual maintenance cost after reengineering
P_5 = Predicted annual operations cost after reengineering
P_6 = Predicted annual business value after reengineering
P_7 = Estimated reengineering costs
P_8 = Estimated reengineering calendar time
P_9 = Reengineering risk factor (P_9 = 1.0 is nominal)
L = Expected life of the system

The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times \text{L} \tag{36.1}$$

The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{36.2}$$

Using the costs presented in Equations (36.1) and (36.2), the overall benefit of reengineering can be computed as

$$\text{Cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \tag{36.3}$$

The cost-benefit analysis presented in these equations can be performed for all high-priority applications identified during inventory analysis (Section 36.4.2). Those applications that show the highest cost-benefit can be targeted for reengineering, while work on others can be postponed until resources are available.

36.10 Summary

Software maintenance and support are ongoing activities that occur throughout the life cycle of an application. During these activities, defects are corrected, applications are adapted to a changing operational or business environment, enhancements are implemented at the request of stakeholders, and users are supported as they integrate an application into their personal or business workflow.

Reengineering occurs at two different levels of abstraction. At the business level, reengineering focuses on the business process with the intent of making changes to improve competitiveness in some area of the business. At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality.

Business process reengineering defines business goals; identifies and evaluates existing business processes (in the context of defined goals); specifies and designs revised processes; and prototypes, refines, and instantiates them within a business. BPR has a focus that extends beyond software. The result of BPR is often the definition of ways in which information technologies can better support the business.

Software reengineering encompasses a series of activities that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability—programs that will be viable well into the twenty-first century.

The cost-benefit of reengineering can be determined quantitatively. The cost of the status quo, that is, the cost associated with ongoing support and maintenance of an existing application, is compared to the projected costs of reengineering and the resultant reduction in maintenance and support costs. In almost every case in which a program has a long life and currently exhibits poor maintainability or supportability, reengineering represents a cost-effective business strategy.

Problems and Points to Ponder

36.1. Consider any job that you've held in the last five years. Describe the business process in which you played a part. Use the BPR model described in Section 36.4.2 to recommend changes to the process in an effort to make it more efficient.

36.2. Do some research on the efficacy of business process reengineering. Present pro and con arguments for this approach.

36.3. Your instructor will select one of the programs that everyone in the class has developed during this course. Exchange your program randomly with someone else in the class. Do not explain or walk through the program. Now, implement an enhancement (specified by your instructor) in the program you have received.

a. Perform all software engineering tasks including a brief walkthrough (but not with the author of the program).
b. Keep careful track of all errors encountered during testing.
c. Discuss your experiences in class.

36.4. Explore the inventory analysis checklist presented at the SEPA website and attempt to develop a quantitative software rating system that could be applied to existing programs in an effort to pick candidate programs for reengineering. Your system should extend beyond the economic analysis presented in Section 36.9.

36.5. Suggest alternatives to paper and ink or conventional electronic documentation that could serve as the basis for document restructuring. (Hint: Think of new descriptive technologies that could be used to communicate the intent of the software.)

36.6. Some people believe that artificial intelligence technology will increase the abstraction level of the reverse engineering process. Do some research on this subject (i.e., the use of AI for reverse engineering), and write a brief paper that takes a stand on this point.

36.7. Why is completeness difficult to achieve as abstraction level increases?

36.8. Why must interactivity increase if completeness is to increase?

36.9. Using information obtained via the Web, present characteristics of three reverse engineering tools to your class.

36.10. There is a subtle difference between restructuring and forward engineering. What is it?

36.11. Research the literature and/or Internet sources to find one or more papers that discuss case studies of mainframe to client-server reengineering. Present a summary.

36.12. How would you determine P_4 through P_7 in the cost-benefit model presented in Section 36.9?

Further Readings and Information Sources

It is ironic that software maintenance and support represent the most costly activities in the life of an application, and yet, fewer books have been written about maintenance and support than any other major software engineering topics. Among recent additions to the literature are books by Reifer (*Software Maintenance Success Recipes,* Auerbach, 2011), Jarzabek (*Effective Software Maintenance and Evolution,* Auerbach, 2007), Grubb and Takang *(Software Maintenance: Concepts and Practice,* World Scientific Publishing Co., 2nd ed., 2003), and Pigoski (*Practical Software Maintenance,* Wiley, 1996). These books cover basic maintenance and support practices and present useful management guidance. Maintenance techniques that focus on client-server environments are discussed by Schneberger (*Client/Server Software Maintenance,* McGraw-Hill, 1997). Current research in "software evolution" is presented in an anthology edited by Mens and Demeyer (*Software Evolution,* Springer, 2008).

Like many hot topics in the business community, the hype surrounding business process reengineering has given way to a more pragmatic view of the subject. Hammer and Champy (*Reengineering the Corporation*, HarperBusiness, revised edition, 2003) precipitated early interest with their best-selling book. Other books by Jacka and Keller (*Business Process Mapping: Improving Customer Satisfaction,* 2nd ed., Wiley, 2009), Sharp and McDermott (*Workflow Modeling,* 2nd ed., Artech House, 2008), Andersen (*Business Process Improvement Toolbox,* 2nd ed., American Society for Quality, 2007), Smith and Fingar [*Business Process Management (BPM): The Third Wave,* Meghan-Kiffer Press, 2003], and Harrington et al. (*Business Process Improvement Workbook,* McGraw-Hill, 1997) present case studies and detailed guidelines for BPR.

Abfalter (*Software Reengineering,* VDM Verlag, 2008), Fong (*Information Systems Reengineering and Integration,* Springer, 2006) describes database conversion techniques, reverse engineering, and forward engineering as they are applied for major information systems. Nierstrasz and his colleagues (*Object Oriented Reengineering Patterns,* Square Bracket Associates, 2009) provides a patterns-based view of how to refactor and/or reengineer OO systems. Secord and his colleagues (*Modernizing Legacy Systems,* Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies,* Prentice Hall, 2002), Valenti (*Successful Software Reengineering,* IRM Press, 2002), and Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) focus on strategies and practices for reengineering at a technical level. Miller (*Reengineering Legacy Software Systems,* Digital Press, 1998) "provides a framework for keeping application systems synchronized with business strategies and technology changes."

Cameron (*Reengineering Business for Success in the Internet Age,* Computer Technology Research, 2000) and Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies,* Prentice Hall, 1997) provide worthwhile guidance for organizations that want to transform legacy systems into a Web-based environment. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems,* Prentice Hall, 1996) discusses the bridge between BPR and information technology. Aiken (*Data Reverse Engineering,* McGraw-Hill, 1996) discusses how to reclaim, reorganize, and reuse organizational data. Arnold (*Software Reengineering,* IEEE Computer Society Press, 1993) has put together an excellent anthology of early papers that focus on software reengineering technologies.

A wide variety of information sources on software reengineering is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

PART

Five

Advanced Topics

In this part of *Software Engineering: A Practitioner's Approach,* we consider a number of advanced topics that will extend your understanding of software engineering. The following questions are addressed in the chapters that follow:

- What is software process improvement and how can it be used to improve the state of software engineering practice?
- What emerging trends can be expected to have a significant influence on software engineering practice in the next decade?
- What is the road ahead for software engineers?

Once these questions are answered, you'll understand topics that may have a profound impact on software engineering in the years to come.

CHAPTER

37 Software Process Improvement

Key Concepts

assessment..... 823
CMMI..... 828
education and training 825
evaluation 826
installation/migration 826
justification 825
maturity models .. 821
people CMM..... 832
return on investment..... 834
risk management . 827
selection..... 825
software process improvement (SPI)
- applicability... 822
- definition of... 819
- frameworks... 819
- process 823

Long before the phrase "software process improvement" was widely used, RSP worked with major corporations in an attempt to improve the state of their software engineering practices. As a consequence of his experiences, he wrote a book entitled *Making Software Engineering Happen* [Pre88]. In the preface of that book he made the following comment:

> For the past ten years I have had the opportunity to help a number of large companies implement software engineering practices. The job is difficult and rarely goes as smoothly as one might like—but when it succeeds, the results are profound. Software projects are more likely to be completed on time. Communication between all constituencies involved in software development is improved. The level of confusion and chaos that is often prevalent for large software projects is reduced substantially. The number of errors encountered by the customer drops substantially. The credibility of the software organization increases. And management has one less problem to worry about.
>
> But all is not sweetness and light. Many companies attempt to implement software engineering practice and give up in frustration. Others do it half-way and never see the benefits noted above. Still others do it in a heavy-handed fashion that results in open rebellion among technical staff and managers and subsequent loss of morale.

Quick Look

What is it? Software process improvement encompasses a set of activities that will lead to a better software process and, as a consequence, higher quality software delivered in a more timely manner.

Who does it? The people who champion SPI come from three groups: technical managers, software engineers, and individuals who have quality assurance responsibility.

Why is it important? Some software organizations have little more than an ad hoc software process. As they work to improve their software engineering practices, they must address weaknesses in their existing process and work to improve their approach to software work.

What are the steps? The approach to SPI is iterative and continuous, but it can be viewed in five steps: (1) assessment of the current software process, (2) education and training of practitioners and managers, (3) selection and justification of process elements, software engineering methods, and tools, (4) implementation of the SPI plan, and (5) evaluation and tuning based on the results of the plan.

What is the work product? Although there are many intermediate SPI work products, the end result is an improved software process that leads to higher quality software.

How do I ensure that I've done it right? The software your organization produces will be delivered with fewer defects, rework at each stage of the software process will be reduced, and on-time delivery will become much more likely.

Although those words were written almost three decades ago, they remain equally true today.

Today, virtually every major software engineering organization has attempted to "make software engineering happen." Some have implemented individual practices that have helped to improve the quality of the product they build and the timeliness of their delivery. Others have established a "mature" software process that guides their technical and project management activities. But others continue to struggle. Their practices are hit-and-miss, and their process is ad hoc. Occasionally, their work is spectacular, but in the main, each project is an adventure, and no one knows whether it will end badly or well.

So, which of these two cohorts needs software process improvement? The answer (which may surprise you) is *both*. Those that have succeeded in making software engineering happen cannot become complacent. They must work continually to improve their approach to software engineering. And those that struggle must begin their journey down the road toward improvement.

37.1 What Is SPI?

KEY POINT

SPI implies a defined software process, an organizational approach, and a strategy for improvement.

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Because SPI is not free, it must deliver a return on investment. The effort and time that is required to implement an SPI strategy must pay for itself in some measurable way. To do this, the results of improved process and practice must lead to a reduction in software "problems" that cost time and money. It must reduce the number of defects that are delivered to end users, reduce the amount of rework due to quality problems, reduce the costs associated with software maintenance and support (Chapter 36), and reduce the indirect costs that occur when software is delivered late.

Quote:

"Much of the software crisis is self-inflicted, as when a CIO says, 'I'd rather have it wrong than have it late. We can always fix it later.'"

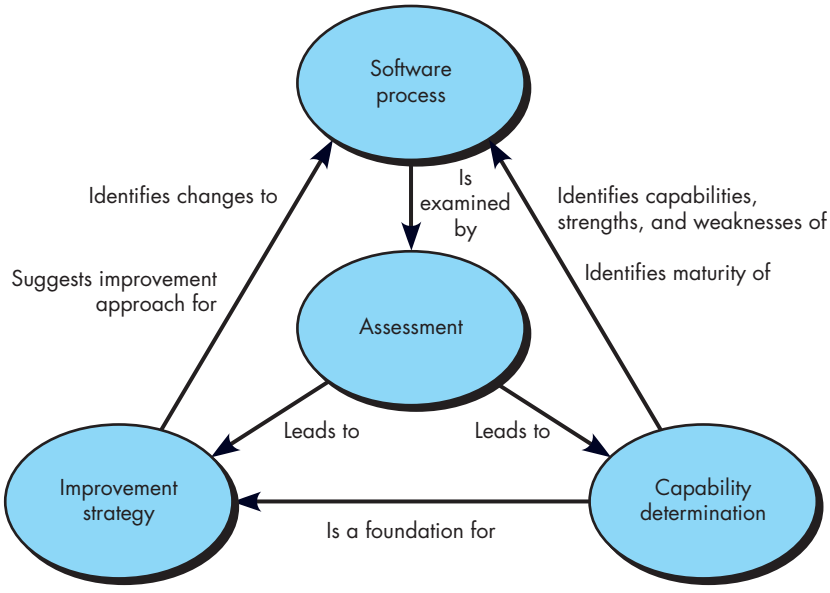
Mark Paulk

37.1.1 Approaches to SPI

Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of SPI frameworks. An *SPI framework* defines (1) a set of characteristics that must be present if an effective software process is to be achieved, (2) a method for assessing whether those characteristics are present, (3) a mechanism for summarizing the results of any assessment, and

FIGURE 37.1

Elements of an SPI framework
Source: Adapted from [Rou02].



(4) a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

An SPI framework assesses the "maturity" of an organization's software process and provides a qualitative indication of a maturity level. In fact, the term "maturity model" (Section 37.1.2) is often applied. In essence, the SPI framework encompasses a maturity model that in turn incorporates a set of process quality indicators that provide an overall measure of the process quality that will lead to product quality.

Figure 37.1 provides an overview of a typical SPI framework. The key elements of the framework and their relationship to one another are shown.

? What groups champion an SPI effort?

You should note that there is no universal SPI framework. In fact, the SPI framework that is chosen by an organization reflects the constituency that is championing the SPI effort. Conradi [Con96] defines six different SPI support constituencies:

> **Quality certifiers.** Process improvement efforts championed by this group focus on the following relationship:
>
> **Quality**(*Process*) ⇒ **Quality**(*Product*)
>
> Their approach is to emphasize assessment methods and to examine a well-defined set of characteristics that allows them to determine whether the process exhibits quality. They are most likely to adopt a process framework such as the CMMI, SPICE, TickIT, or Bootstrap.[1]

1 Each of these SPI frameworks is discussed later in this chapter.

Formalists. This group wants to understand (and when possible, optimize) process workflow. To accomplish this, they use process modeling languages (PMLs) to create a model of the existing process and then design extensions or modifications that will make the process more effective.

Tool advocates. This group insists on a tool-assisted approach to SPI that models workflow and other process characteristics in a manner that can be analyzed for improvement.

Practitioners. This constituency uses a pragmatic approach, "emphasizing mainstream project-, quality- and product management, applying project-level planning and metrics, but with little formal process modeling or enactment support" [Con96].

Reformers. The goal of this group is organizational change that might lead to a better software process. They tend to focus more on human issues (Section 37.5) and emphasize measures of human capability and structure.

Ideologists. This group focuses on the suitability of a particular process model for a specific application domain or organizational structure. Rather than typical software process models (e.g., iterative models), ideologists would have a greater interest in a process that would, say, support reuse or reengineering.

As an SPI framework is applied, the sponsoring constituency (regardless of its overall focus) must establish mechanisms to: (1) support technology transition, (2) determine the degree to which an organization is ready to absorb process changes that are proposed, and (3) measure the degree to which changes have been adopted.

37.1.2 Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization. That is, an indication of the quality of the software process, the degree to which practitioners understand and apply the process, and the general state of software engineering practice. This is accomplished using some type of ordinal scale.

KEY POINT
A maturity model defines levels of software process competence and implementation.

For example, the Software Engineering Institute's original *Capability Maturity Model* (Section 37.3) suggests five levels of maturity ranging from *initial* (rudimentary software process) to *optimized* (a process that leads to best practices.[2] Unfortunately, some software organizations exhibit levels of "process immaturity" that undermine any rational attempt at improving software

2 The original CMM has been updated and is discussed in Section 37.3.

engineering practices. Schorsch [Sch06] suggests four levels of organizational immaturity that are often encountered in the real world of software development:

> *Level 0, Negligent*—Failure to allow successful development process to succeed. All problems are perceived to be technical problems. Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process. Reliance on silver pellets.
>
> *Level 1, Obstructive*—Counterproductive processes are imposed. Processes are rigidly defined and adherence to the form is stressed. Ritualistic ceremonies abound. Collective management precludes assigning responsibility. Status quo über alles.
>
> *Level 2, Contemptuous*—Disregard for good software engineering institutionalized. Complete schism between software development activities and software process improvement activities. Complete lack of a training program.
>
> *Level 3, Undermining*—Total neglect of own charter, conscious discrediting of peer organizations software process improvement efforts. Rewarding failure and poor performance.

Schorsch's immaturity levels are toxic for any software organization. If you encounter any one of them, attempts at SPI are doomed to failure.

The overriding question is whether maturity scales, such as the one proposed as part of the CMM, provide any real benefit. We think that they do. A maturity scale provides an easily understood snapshot of process quality that can be used by practitioners and managers as a benchmark from which improvement strategies can be planned.

37.1.3 Is SPI for Everyone?

For many years, SPI was viewed as a "corporate" activity—a euphemism for something that only large companies perform. But today, a significant percentage of all software development is being performed by companies that employ fewer than 100 people. Can a small company initiate SPI activities and do it successfully?

ADVICE

If a specific process model or SPI approach feels like overkill for your organization, it probably is.

There are substantial cultural differences between large software development organizations and small ones. It should come as no surprise that small organizations are more informal, apply fewer standard practices, and tend to be self-organizing. They also tend to pride themselves on the "creativity" of individual members of the software organization, and initially view an SPI framework as overly bureaucratic and ponderous. Yet, process improvement is as important for a small organization as it is for a large one.

Within small organizations the implementation of an SPI framework requires resources that may be in short supply. Managers must allocate people and money to make software engineering happen. Therefore, regardless of the size of the software organization, it's reasonable to consider the business motivation for SPI.

SPI will be approved and implemented only after its proponents demonstrate *financial leverage* [Bir98]. Financial leverage is demonstrated by examining technical benefits (e.g., fewer defects delivered to the field, reduced rework, lower maintenance costs, or more rapid time-to-market) and translating them into dollars. In essence, you must show a realistic return on investment (Section 37.7) for SPI costs.

37.2 The SPI Process

The hard part of SPI isn't the definition of characteristics that define a high-quality software process or the creation of a process maturity model. Those things are relatively easy. Rather, the hard part is establishing a consensus for initiating SPI and defining an ongoing strategy for implementing it across a software organization.

The Software Engineering Institute has developed IDEAL— "an organizational improvement model that serves as a road map for initiating, planning, and implementing improvement actions" [SEI08]. IDEAL is representative of many process models for SPI, defining five distinct activities— initiating, diagnosing, establishing, acting, and learning—that guide an organization through SPI activities.

In this book, we present a somewhat different road map for SPI, based on the process model for SPI originally proposed in [Pre88]. It applies a commonsense philosophy that requires an organization to (1) look in the mirror, (2) then get smarter so it can make intelligent choices, (3) select the process model (and related technology elements) that best meets its needs, (4) instantiate the model into its operating environment and its culture, and (5) evaluate what has been done. These five activities (discussed in the subsections[3] that follow) are applied in an iterative (cyclical) manner in an effort to foster continuous process improvement.

37.2.1 Assessment and Gap Analysis

Any attempt to improve your current software process without first assessing the efficacy of current framework activities and associated software engineering practices would be like starting on a long journey to a new location with no idea where you are starting from. You'd depart with great flourish, wander around trying to get your bearings, expend lots of energy and endure large doses of frustration, and likely, decide you really didn't want to travel anyway. Stated simply, before you begin any journey, it's a good idea to know precisely where you are.

3 Some of the content in these sections has been adapted from [Pre88] with permission.

ADVICE

Be sure to understand your strengths as well as your weaknesses. If you're smart, you'll build on the strengths.

The first road map activity, called *assessment,* allows you to get your bearings. The intent of assessment is to uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process.

Assessment examines a wide range of actions and tasks that will lead to a high-quality process. For example, regardless of the process model that is chosen, the software organization must establish generic mechanisms such as defined approaches for customer communication; established methods for representing user requirements; a project management framework that includes scoping, estimation, scheduling, and project tracking; risk analysis methods; change management procedures; quality assurance and control activities including reviews; and many others. Each is considered within the context of the framework activities (Chapter 3) that have been established, and is each assessed to determine whether all of the following questions have been addressed:

- Is the objective of the activity clearly defined?
- Are work products required as input and produced as output identified and described?
- Are the work tasks to be performed clearly described?
- Are the people who must perform the activity identified by role?
- Have entry and exit criteria been established?
- Have metrics for the activity been established?
- Are tools available to support the activity?
- Is there an explicit training program that addresses the activity?
- Is the activity performed uniformly for all projects?

Although the questions noted imply a *yes* or *no* answer, the role of assessment is to look behind the answer to determine whether the activity in question is being performed in a manner that would conform to best practice.

As the process assessment is conducted, you (or those who have been hired to perform the assessment) should also focus on the following issues:

Consistency. Are important activities, actions, and tasks applied consistently across all software projects and by all software teams?

Sophistication. Are management and technical actions performed with a level of sophistication that implies a thorough understanding of best practice?

Acceptance. Is the software process and software engineering practice widely accepted by management and technical staff?

Commitment. Has management committed the resources required to achieve consistency, sophistication, and acceptance?

The difference between local application and best practice represents a "gap" that offers opportunities for improvement. The degree to which consistency, sophistication, acceptance, and commitment are achieved indicates the amount of cultural change that will be required to achieve meaningful improvement.

37.2.2 Education and Training

ADVICE

Try to provide "just-in-time" training targeted to the real needs of a software team.

Although few software people question the benefit of an agile, organized software process or solid software engineering practices, many practitioners and managers do not know enough about either subject.[4] As a consequence, inaccurate perceptions of process and practice lead to inappropriate decisions when an SPI framework is introduced. It follows that a key element of any SPI strategy is education and training for practitioners, technical managers, and more senior managers who have direct contact with the software organization. Three types of education and training should be conducted: generic software engineering concepts and methods, specific technology and tools, and communication and quality-oriented topics. In a modern context, education and training can be delivered in a variety of different ways. Everything from podcasts, to short YouTube videos, to more comprehensive Internet-based training (e.g., [QAI08], to DVDs, to classroom courses can be offered as part of an SPI strategy.

37.2.3 Selection and Justification

ADVICE

As you make your choices, be sure to consider the culture of your organization and the level of acceptance that each choice will likely elicit.

Once the initial assessment activity[5] has been completed and education has begun, a software organization should begin to make choices. These choices occur during a *selection and justification activity* in which process characteristics and specific software engineering methods and tools are chosen to populate the software process.

First, you should choose the process model (Chapters 3–5) that best fits your organization, its stakeholders, and the software that you build. You should decide which of the set of framework activities will be applied, the major work products that will be produced, and the quality assurance checkpoints that will enable your team to assess progress. If the SPI assessment activity indicates that you have specific weaknesses (e.g., you have no formal SQA functions), you should focus attention on process characteristics that will address these weaknesses directly.

Next, develop an adaptable work breakdown for each framework activity (e.g., modeling), defining the task set that would be applied for a typical project. You

4 If you've spent time reading this book, you won't be one of them!

5 In actuality, assessment is an ongoing activity. It is conducted periodically in an effort to determine whether the SPI strategy has achieved its immediate goals and to set the stage for future improvement.

should also consider the software engineering methods that can be applied to achieve these tasks. As choices are made, education and training should be coordinated to ensure that understanding is reinforced.

Ideally, everyone works together to select various process and technology elements and moves smoothly toward the installation or migration activity (Section 37.2.4). In reality, selection can be a rocky road. It is often difficult to achieve consensus among different constituencies. If the criteria for selection are established by committee, people may argue endlessly about whether the criteria are appropriate and whether a choice truly meets the criteria that have been established.

It is true that a bad choice can do more harm than good, but "paralysis by analysis" means that little if any progress occurs and process problems remain. As long as the process characteristic or technology element has a good chance at meeting an organization's needs, it's sometimes better to pull the trigger and make a choice, rather than waiting for the perfect solution.

37.2.4 Installation/Migration

Installation is the first point at which a software organization feels the effects of changes implemented as a consequence of the SPI road map. In some cases, an entirely new process is recommended for an organization. Framework activities, software engineering actions, and individual work tasks must be defined and installed as part of a new software engineering culture. Such changes represent a substantial organizational and technological transition and must be managed very carefully.

In other cases, changes associated with SPI are relatively minor, representing small, but meaningful modifications to an existing process model. Such changes are often referred to as *process migration*. Today, many software organizations have a "process" in place. The problem is that it doesn't work in an effective manner. Therefore, an incremental *migration* from one process (that doesn't work as well as desired) to another process is a more effective strategy.

Installation and migration are actually *software process redesign* (SPR) activities. Scacchi [Sca00] states that "SPR is concerned with identification, application, and refinement of new ways to dramatically improve and transform software processes." When a formal approach to SPR is initiated, three different process models are considered: (1) the existing ("as is") process, (2) a transitional ("here to there") process, and the target ("to be") process. If the target process is significantly different from the existing process, the only rational approach to installation is an incremental strategy in which the transitional process is implemented in steps. The transitional process provides a series of way-points that enable the software organization's culture to adapt to small changes over a period of time.

37.2.5 Evaluation

Although it is listed as the last activity in the SPI road map, *evaluation* occurs throughout SPI. The evaluation activity assesses the degree to which changes have been instantiated and adopted, the degree to which such changes result in better software quality or other tangible process benefits, and the overall status of the process and the organizational culture as SPI activities proceed.

Both qualitative factors and quantitative metrics are considered during the evaluation activity. From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes. Quantitative metrics (Chapter 32) are collected from projects that have used the transitional or "to be" process and compared with similar metrics that were collected for projects that were conducted under the "as is" process.

37.2.6 Risk Management for SPI

KEY POINT

SPI often fails because risks were not properly considered and no contingency planning occurred.

SPI is a risky undertaking. In fact, more than half of all SPI efforts end in failure. The reasons for failure vary greatly and are organizationally specific. Among the most common risks are: a lack of management support, cultural resistance by technical staff, a poorly planned SPI strategy, an overly formal approach to SPI, selection of an inappropriate process, a lack of buy-in by key stakeholders, an inadequate budget, a lack of staff training, organizational instability, and a myriad of other factors. The role of those chartered with the responsibility for SPI is to analyze likely risks and develop an internal strategy for mitigating them.

A software organization should manage risk at three key points in the SPI process [Sta97b]: prior to the initiation of the SPI road map, during the execution of SPI activities (assessment, education, selection, installation), and during the evaluation activity that follows the instantiation of some process characteristic. In general, the following categories [Sta97b] can be identified for SPI risk factors: budget and cost, content and deliverables, culture, maintenance of SPI deliverables, mission and goals, organizational management, organizational stability, process stakeholders, schedule for SPI development, SPI development environment, SPI development process, SPI project management, and SPI staff.

Within each category, a number of generic risk factors can be identified. For example, the organizational culture has a strong bearing on risk. The following generic risk factors[6] can be defined for the culture category [Sta97b]:

- Attitude toward change, based on prior efforts to change
- Experience with quality programs, level of success

6 Risk factors for each of the risk categories noted in this section can be found in [Sta97b].

- Action orientation for solving problems versus political struggles
- Use of facts to manage the organization and business
- Patience with change; ability to spend time socializing
- Tools orientation—expectation that tools can solve the problems
- Level of "planfulness"—ability of organization to plan
- Ability of organization members to participate with various levels of organization openly at meetings
- Ability of organization members to manage meetings effectively
- Level of experience in organization with defined processes

Using the risk factors and generic attributes as a guide, a risk table (Chapter 35) can be developed to isolate those risks that warrant further management attention.

37.3 The CMMI

The original CMM was developed and upgraded by the Software Engineering Institute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability Maturity Model Integration* (CMMI) [CMM07], a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure 37.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

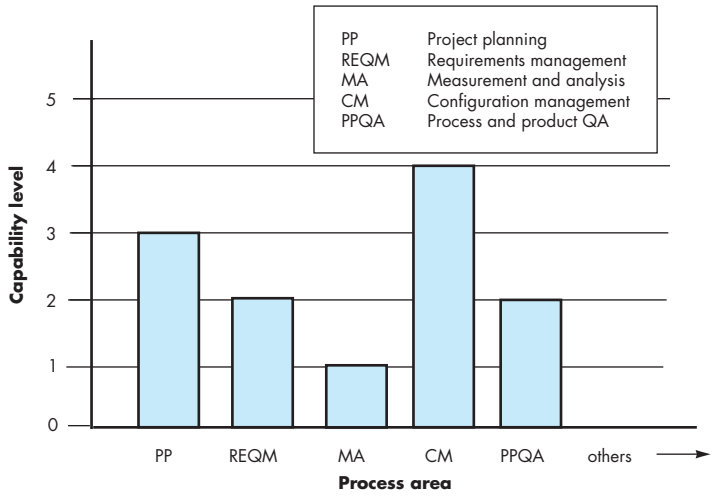
Level 0: *Incomplete*— The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

Level 1: *Performed*— All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: *Managed*— All capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in

FIGURE 37.2

CMMI Process Area Capability Profile
Source: [Phi02].



the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM07].

ADVICE

Every organization should strive to achieve the intent of the CMMI. However, implementing every aspect of the model may be overkill in your situation.

Level 3: *Defined*— All capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" [CMM07].

Level 4: *Quantitatively managed*— All capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM07].

Level 5: *Optimized*— All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

WebRef
Complete information as well as a downloadable version of the CMMI can be obtained at **http://cmmiinstitute.com/resources/**.

For example, **project planning** is one of eight process areas defined by the CMMI for "project management" category.[7] The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are [CMM07]:

SG 1 Establish Estimates

SP 1.1-1 Estimate the Scope of the Project

SP 1.2-1 Establish Estimates of Work Product and Task Attributes

SP 1.3-1 Define Project Life Cycle

SP 1.4-1 Determine Estimates of Effort and Cost

SG 2 Develop a Project Plan

SP 2.1-1 Establish the Budget and Schedule

SP 2.2-1 Identify Project Risks

SP 2.3-1 Plan for Data Management

SP 2.4-1 Plan for Project Resources

SP 2.5-1 Plan for Needed Knowledge and Skills

SP 2.6-1 Plan Stakeholder Involvement

SP 2.7-1 Establish the Project Plan

SG 3 Obtain Commitment to the Plan

SP 3.1-1 Review Plans That Affect the Project

SP 3.2-1 Reconcile Work and Resource Levels

SP 3.3-1 Obtain Plan Commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved.

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity levels and process areas is shown in Figure 37.3.

7 Other process areas defined for "project management" include: project monitoring and control, supplier agreement management, integrated project management for IPPD, risk management, integrated teaming, integrated supplier management, and quantitative project management.

FIGURE 37.3

Process areas required to achieve a maturity level
Source: [Phi02].

Level	Focus	Process Areas
Optimizing	***Continuous process improvement***	**Organizational innovation and deployment** **Causal analysis and resolution**
Quantitatively managed	***Quantitative management***	**Organizational process performance** **Quantitative project management**
Defined	***Process standardization***	**Requirements development** **Technical solution** **Product integration** **Verification** **Validation** **Organizational process focus** **Organizational process definition** **Organizational training** **Integrated project management** **Integrated supplier management** **Risk management** **Decision analysis and resolution** **Organizational environment for integration** **Integrated teaming**
Managed	***Basic project management***	**Requirements management** **Project planning** **Project monitoring and control** **Supplier agreement management** **Measurement and analysis** **Process and product quality assurance** **Configuration management**
Performed		

INFO

The CMMI—Should We or Shouldn't We?

The CMMI is a process meta-model. It defines (in 700+ pages) the process characteristics that should exist if an organization wants to establish a software process that is complete. The question that has been debated for almost two decades is: "Is the CMMI overkill?" Like most things in life (and in software), the answer is not a simple yes or no.

The spirit of the CMMI should always be adopted. At the risk of oversimplification, it argues that software development must be taken seriously—it must be planned thoroughly, it must be controlled uniformly, it must be tracked accurately, and it must be conducted professionally. It must focus on the needs of project stakeholders, the skills of the software engineers, and the quality of the end product. No one would argue with these ideas.

The detailed requirements of the CMMI should be seriously considered if an organization builds large complex systems that involve dozens or hundreds of people over many months or years. It may be that the CMMI is "just right" in such situations, if the organizational culture is amenable to standard process models and management is committed to making it a success. However, in other situations, the CMMI may simply be too much for an organization to successfully assimilate. Does this mean that the CMMI is "bad" or "overly bureaucratic" or "old fashioned?" No . . . it does not. It simply means

that what is right for one organizational culture may not be right for another.

The CMMI is a significant achievement in software engineering. It provides a comprehensive discussion of the activities and actions that should be present when an organization builds computer software. Even if a software organization chooses not to adopt its details, every software team should embrace its spirit and gain insight from its discussion of software engineering process and practice.

37.4 The People CMM

KEY POINT

The People CMM suggests practices that improve the workforce competence and culture.

A software process, no matter how well conceived, will not succeed without talented, motivated software people. The *People Capability Maturity Model* "is a road map for implementing workforce practices that continuously improve the capability of an organization's workforce" [Cur01]. Developed in the mid-1990s and refined over the intervening years, the goal of the People CMM is to encourage continuous improvement of generic workforce knowledge (called "core competencies"), specific software engineering and project management skills (called "workforce competencies"), and process-related abilities.

Like the CMM, CMMI, and related SPI frameworks, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes. These maturity levels [CMM08] are tied to the existence (within an organization) of a set of key process areas (KPAs). An overview of organizational levels and related KPAs is shown in Figure 37.4.

The People CMM complements any SPI framework by encouraging an organization to nurture and improve its most important asset—its people. As important, it establishes a workforce atmosphere that enables a software organization to "attract, develop, and retain outstanding talent" [CMM08].

37.5 Other SPI Frameworks

Although the SEI's CMM and CMMI are the most widely applied SPI frameworks, a number of alternatives[8] have been proposed and are in use. We provide a brief overview of these frameworks in the paragraphs that follow.[9]

8 It's reasonable to argue that some of these frameworks are not so much "alternatives" as they are complementary approaches to SPI. A comprehensive table of many more SPI frameworks can be found at **http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.4787&rep=rep1&type=pdf**

9 If you have further interest, a wide array of print and Web-based resources is available for each.

FIGURE 37.4

Process areas for the People CMM

Level	Focus	Process Areas
Optimized	***Continuous improvement***	**Continuous workforce innovation** **Organizational performance alignment** **Continuous capability improvement**
Predictable	***Quantifies and manages knowledge, skills, and abilities***	**Mentoring** **Organizational capability management** **Quantitative performance management** **Competency-based assets** **Empowered workgroups** **Competency integration**
Defined	***Identifies and develops knowledge, skills, and abilities***	**Participatory culture** **Workgroup development** **Competency-based practices** **Career development** **Competency development** **Workforce planning** **Competency analysis**
Managed	***Repeatable, basic people management practices***	**Compensation** **Training and development** **Performance management** **Work environment** **Communication and co-ordination** **Staffing**
Initial	***Inconsistent practices***	

? In addition to the CMM, are there other SPI frameworks that we might consider?

SPICE. The SPICE (*Software Process Improvement and Capability dEtermination*) model [SPI99] provides an SPI assessment framework that is compliant with ISO 15504:2003 and ISO 12207. The SPICE document suite [SDS08] presents a complete SPI framework including a model for process management, guidelines for conducting an assessment and rating the process under consideration, construction, selection, and use of assessment instruments and tools, and training for assessors.

Bootstrap. The *Bootstrap* SPI framework "has been developed to ensure conformance with the emerging ISO standard for software process assessment and improvement (SPICE) and to align the methodology with ISO 12207" [Boo06]. The objective of Bootstrap is to evaluate a software process using a set of software engineering best practices as a basis for assessment. Like the CMMI, Bootstrap provides a process maturity level using the results of questionnaires that gather information about the "as is" software process and software projects. SPI guidelines are based on maturity level and organizational goals.

PSP and TSP. Although SPI is generally characterized as an organizational activity, there is no reason why process improvement cannot be conducted at an

Quote:

"Software organizations have exhibited significant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

individual or team level. Both PSP and TSP (Chapter 4 emphasize the need to continuously collect data about the work that is being performed and to use that data to develop strategies for improvement. Watts Humphrey, (Hum97], [Hum00]) the developer of both methods, comments:

> The PSP [and TSP] will show you how to plan and track your work and how to consistently produce high quality software. Using PSP [and TSP] will give you the data that show the effectiveness of your work and identify your strengths and weaknesses . . . To have a successful and rewarding career, you need to know your skills and abilities, strive to improve them, and capitalize on your unique talents in the work you do.

TickIT. The Ticket auditing method [Tic05] ensures compliance with *ISO 9001:2000 for Software*—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

The underlying strategy suggested by ISO 9001:2000 is described in the following manner [ISO00]:

> ISO 9001:2000 stresses the importance for an organization to identify, implement, manage and continually improve the effectiveness of the processes that are necessary for the quality management system, and to manage the interactions of these processes in order to achieve the organization's objectives . . . Process effectiveness and efficiency can be assessed through internal or external review processes and be evaluated on a maturity scale.

WebRef

An excellent summary of ISO 9001: 2008 can be found at **http://praxiom.com/iso-9001.htm.**

ISO 9001:2008 has adopted a "plan-do-check-act" cycle that is applied to the quality management elements of a software project. Within a software context, "plan" establishes the process objectives, activities, and tasks necessary to achieve high-quality software and resultant customer satisfaction. "Do" implements the software process (including both framework and umbrella activities). "Check" monitors and measures the process to ensure that all requirements established for quality management have been achieved. "Act" initiates software process improvement activities that continually work to improve the process. TickIt can be used throughout the "plan-do-check-act" cycle to ensure that SPI progress is being made. TickIT auditors assess the application of the cycle as a precursor to ISO 9001:2008 certification. For a detailed discussion of ISO 9001:2008 and TickIT you should examine [Ant06], [Tri05], or [Sch03].

37.6 SPI Return on Investment

SPI is hard work and requires substantial investment of dollars and people. Managers who approve the budget and resources for SPI will invariably ask the question: "How do I know that we'll achieve a reasonable return for the money we're spending?"

At a qualitative level, proponents of SPI argue that an improved software process will lead to improved software quality. They contend that improved process will result in the implementation of better-quality filters (resulting in fewer propagated defects), better control of change (resulting in less project chaos), and less technical rework (resulting in lower cost and better time-to-market). But can these qualitative benefits be translated into quantitative results? The classic return on investment (ROI) equation is:

$$\text{ROI} = \left[\frac{[\Sigma\ (benefits) - \Sigma\ (costs)]}{\Sigma\ (costs)}\right] \times 100\%$$

where

benefits include the cost savings associated with higher product quality (fewer defects), less rework, reduced effort associated with changes, and the income that accrues from shorter time-to-market.

costs include both direct SPI costs (e.g., training, measurement) and indirect costs associated with greater emphasis on quality control and change management activities and more rigorous application of software engineering methods (e.g., the creation of a design model).

In the real world, these quantitative benefits and costs are sometimes difficult to measure with accuracy, and all are open to interpretation. But that doesn't mean that a software organization should conduct an SPI program without careful analysis of the costs and benefits that accrue. A comprehensive treatment of ROI for SPI can be found in a unique book by David Rico [Ric04].

37.7 SPI Trends

Over the past 25 years, many companies have attempted to improve their software engineering practices by applying an SPI framework to effect organizational change and technology transition. As we noted earlier in this chapter, over half fail in this endeavor. Regardless of success or failure, all spend significant amounts of money. David Rico [Ric04] reports that a typical application of an SPI framework such as the SEI CMM can cost between $25,000 and $70,000 per person and take years to complete! It should come as no surprise that the future of SPI should emphasize a less costly and time-consuming approach.

To be effective in the twenty-first-century world of software development, future SPI frameworks must become significantly more agile. Rather than an organizational focus (which can take years to complete successfully), contemporary SPI efforts should focus on the project level, working to improve a team process in weeks, not months or years. To achieve meaningful results (even at the project level) in a short time frame, complex framework models may give way to simpler models. Rather than dozens of key practices and hundreds of supplementary

practices, an agile SPI framework should emphasize only a few pivotal practices (e.g., analogous to the framework activities discussed throughout this book).

Any attempt at SPI demands a knowledgeable workforce, but education and training expenses can be expensive and should be minimized (and streamlined). Rather than classroom courses (expensive and time-consuming), future SPI efforts should rely on Web-based training that is targeted at pivotal practices. Rather than far-reaching attempts to change organizational culture (with all of the political perils that ensue), cultural change should occur as it does in the real world, one small group at a time until a tipping point is reached.

The SPI work of the past two decades has significant merit. The frameworks and models that have been developed represent substantial intellectual assets for the software engineering community. But like all things, these assets guide future attempts at SPI not by becoming a recurring dogma, but by serving as the basis for better, simpler, and more agile SPI models.

37.8 Summary

A software process improvement framework defines the characteristics that must be present if an effective software process is to be achieved, an assessment method that helps determine whether those characteristics are present, and a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing. Regardless of the constituency that sponsors SPI, the goal is to improve process quality and, as a consequence, improve software quality and timeliness.

A process maturity model provides an overall indication of the "process maturity" exhibited by a software organization. It provides a qualitative feel for the relative effectiveness of the software process that is currently being used.

The SPI road map begins with assessment—a series of evaluation activities that uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process. As a consequence of assessment, a software organization can develop an overall SPI plan.

One of the key elements of any SPI plan is education and training, an activity that focuses on improving the knowledge level of managers and practitioners. Once staff becomes well versed in current software technologies, selection and justification commence. These tasks lead to choices about the architecture of the software process, the methods that populate it, and the tools that support it. Installation and evaluation are SPI activities that instantiate process changes and assess their efficacy and impact.

To successfully improve its software process, an organization must exhibit the following characteristics: management commitment and support for SPI, staff involvement throughout the SPI process, process integration into the overall

organizational culture, an SPI strategy that has been customized for local needs, and solid management of the SPI project.

A number of SPI frameworks are in use today. The SEI's CMM and CMMI are widely used. The People CMM has been customized to assess the quality of the organizational culture and the people who populate it. SPICE, Bootstrap, PSP, TSP, and TickIT are additional frameworks that can lead to effective SPI.

SPI is hard work that requires substantial investment of dollars and people. To ensure that a reasonable return on investment is achieved, an organization must measure the costs associated with SPI and the benefits that can be directly attributed to it.

Problems and Points to Ponder

37.1. Why is it that software organizations often struggle when they embark on an effort to improve local software process?

37.2. Describe the concept of "process maturity" in your own words.

37.3. Do some research (check the SEI website) and determine the process maturity distribution for software organizations in the United States and worldwide.

37.4. You work for a very small software organization—only 11 people are involved in developing software. Is SPI for you? Explain your answer.

37.5. Assessment is analogous to an annual physical exam. Using a physical exam as a metaphor, describe the SPI assessment activity.

37.6. What is the difference between an "as is" process, a "here to there" process, and a "to be" process?

37.7. How is risk management applied within the context of SPI?

37.8. Select one of the critical success factors noted in Section 37.2.7. Do some research and write a brief paper on how it can be achieved.

37.9. Do some research and explain how the CMMI differs from its predecessor, the CMM.

37.10. Select one of the SPI frameworks discussed in Section 37.5, and write a brief paper describing it in more detail.

Further Readings and Information Sources

One of the most readily accessible and comprehensive resources for information on SPI has been developed by the Software Engineering Institute and is available at **www.sei.cmu.edu** and **www.cmmiinstitute.com**. The SEI websites contain hundreds of papers, studies, and detailed SPI framework descriptions.

Over the past few years, a number of worthwhile books have been added to a broad literature developed during the past two decades. Chrissis and her colleagues (*CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed., Addison-Wesley, 2011) and McMahin (*Integrating CMMI and Agile Development,* Addison-Wesley, 2010) discuss the application of CCMI in modern software development. Land (*Jumpstart CMM/CMMI Software Process Improvements,* Wiley-IEEE Computer Society, 2007) melds the requirements defined as part of the SEI CMM and CMMI with IEEE software engineering standards with an emphasis on the intersection of process and practice. Micklewright (*Lean*

ISO 9001, ASQ Quality Press, 2010) and Cianfrani and his colleagues (*ISO 9001:2008* Explained, 3rd ed., ASQ Quality Press, 2009) describe the meaning and intent of ISO 9001:2008. Mutafelija and Stromberg (*Systematic Process Improvement Using ISO 9001:2000 and CMMI,* Artech House Publishers, 2007) discuss both the ISO 9001:2000 and CMMI SPI frameworks and the "synergy" between them. Conradi and his colleagues (*Software Process Improvement: Results and Experience from the Field,* Springer, 2006) presents the results of a series of case studies and experiments related to SPI.

McKay and Black (*Improving the Software Process,* RBCS, 2012) provide a detailed look at SPI approaches and trends. Fauzi and his colleagues (*Software Process Improvement: Approaches and Tools for Practical Development,* IGI Global, 2011), Van Loon (*Process Assessment and Improvement: A Practical Guide for Managers, Quality Professionals and Assessors,* Springer, 2006) discusses SPI within the context of ISO/IEC 15504. Watts Humphrey (*PSP,* Addison-Wesley, 2005, and *TSP,* Addison-Wesley, 2005) addresses his Personal Team Process SPI framework and his Team Software Process SPI framework in two separate books. Fantina (*Practical Software Process Improvement,* Artech House Publishers, 2004) provides pragmatic how-to guidance with an emphasis on CMMI/CMM.

A wide variety of information sources on software process improvement is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman**.

CHAPTER

38

Emerging Trends in Software Engineering

Key Concepts

building blocks . . . 847
collaborative development. 852
complexity 845
emergent requirements 846
hype cycle. 842
innovation life cycle. 840
model-driven development. 853
open source 848
open-world software 846
postmodern design . 854
requirements engineering. 852
soft trends 843
technology directions 849
technology evolution 840

Throughout the relatively brief history of software engineering, practitioners and researchers have developed an array of process models, technical methods, and automated tools in an effort to foster fundamental change in the way we build computer software. Even though past experience indicates otherwise, there is a tacit desire to find the "silver bullet"—the magic process or transcendent technology that will allow us to build large, complex, software-based systems easily, without confusion, without mistakes, without delay—without the many problems that continue to plague software work.

But history indicates that our quest for the silver bullet appears doomed to failure. New technologies are introduced regularly, hyped as a "solution" to many of the problems software engineers face, and incorporated into projects large and small. Industry pundits stress the importance of these "new" software technologies, the cognoscenti of the software community adopt them with enthusiasm, and ultimately, they do play a role in the software engineering world. But they tend not to meet their promise, and as a consequence, the quest continues.

In past editions of this book (over the past 35 years), we have discussed emerging technologies and their projected impact on software engineering.

Quick Look

What is it? No one can predict the future with absolute certainty. But it is possible to assess trends in the software engineering area and from those trends to suggest possible directions for the technology. That's what we attempt to do in this chapter.

Who does it? Anyone who is willing to spend the time to stay abreast of software engineering issues can try to predict the future direction of the technology.

Why is it important? Why did ancient kings hire soothsayers? Why do major multinational corporations hire consulting firms and think tanks to prepare forecasts? Why does a substantial percentage of the public read horoscopes? We want to know what's coming so we can ready ourselves.

What are the steps? There is no formula for predicting the road ahead. We attempt to do this by collecting data, organizing it to provide useful information, examining subtle associations to extract knowledge, and from this knowledge to suggest probable trends that predict how things will be at some future time.

What is the work product? A view of the near-term future that may or may not be correct.

How do I ensure that I've done it right? Predicting the road ahead is an art, not a science. In fact, it's quite rare when a serious prediction about the future is absolutely right or unequivocally wrong (with the exception, thankfully, of predictions of the end of the world). We look for trends and try to extrapolate them. We can assess the correctness of the extrapolation only as time passes.

test-driven development..... 854
tools 855

Some have been widely adopted, but others never reached their potential. Our conclusion: technologies come and go; the real trends that we should explore are softer. By this we mean that progress in software engineering will be guided by business, organizational, market, and cultural trends. Those trends lead to technology innovation.

In this chapter, we'll mention a few software engineering technology trends, but our primary emphasis will be on some of the business, organizational, market, and cultural trends that may have an important influence on software engineering technology over the next 10 or 20 years.

38.1 Technology Evolution

In a fascinating book that provides a compelling look at how computing (and other related) technologies will evolve, Ray Kurzweil [Kur05] argues that technological evolution is similar to biological evolution, but occurs at a rate that is orders of magnitude faster. Evolution (whether biological or technological) occurs as a result of positive feedback—"the more capable methods resulting from one stage of evolutionary progress are used to create the next stage" [Kur05].

? What are the "big questions" when we consider technology evolution?

The big questions for the 21st century are: (1) How rapidly does a technology evolve? (2) How significant are the effects of positive feedback? (3) How profound will the resultant changes be?

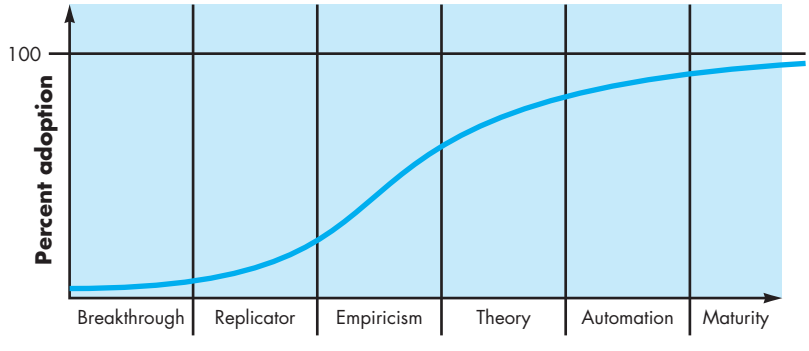
KEY POINT

Computing technology is evolving at an exponential rate, and its growth may soon become explosive.

When a successful new technology is introduced, the initial concept moves through a reasonably predictable "innovation life cycle" [Gai95] illustrated in Figure 38.1. In the *breakthrough* phase, a problem is recognized and repeated attempts at a viable solution are attempted. At some point, a solution shows promise. The initial breakthrough work is reproduced in the *replicator* phase and gains wider usage. *Empiricism* leads to the creation of empirical rules that govern the use of the technology, and repeated success leads to a broader *theory* of usage that is followed by the creation of automated tools during the *automation* phase. Finally, the technology matures and is used widely.

FIGURE 38.1

A technology innovation life cycle



You should note that many research and technology trends never reach maturity. In fact, the vast majority of "promising" technologies in the software engineering domain receive widespread interest for a few years and then fall into niche usage by a dedicated band of adherents. This is not to say that these technologies lack merit, but rather to emphasize that the journey through the innovation life cycle is long and hard.

Kurzweil [Kur05] agrees that computing technologies evolve through an "S-curve" that exhibits relatively slow growth during the technology's formative years, rapid acceleration during its growth period, and then a leveling-off period as the technology reaches its limits. Today, we a rapidly accelerating through the knee of the S-curve for modern computing technologies—at the transition between early growth and the explosive growth that is to follow. The implication is that over the next 20 to 40 years, we will see dramatic (even mind-boggling) changes in computing capability. He suggests that within 20 years, technology evolution will accelerate at an increasingly rapid pace, ultimately leading to an era of nonbiological intelligence that will merge with and extend human intelligence in ways that are fascinating to contemplate.

And all of this, no matter how it evolves, will require software and systems that make our current efforts look infantile by comparison. By the year 2040, a combination of extreme computation, nanotechnology, massively high bandwidth ubiquitous networks, and robotics will lead us into a different world.[1] Software, possibly in forms we cannot yet comprehend, will continue to reside at the core of this new world. Software engineering will not go away.

38.2 Prospects for a True Engineering Discipline

For almost 50 years, many academic researchers and industry professionals have clamored for a true engineering discipline for software. In an important follow-on to her classic 1990 paper the subject, Mary Shaw [Sha09] comments on this continuing quest:

> Engineering disciplines typically evolve from craft practices of a technology, sufficient for local or ad hoc use. When the technology becomes economically significant, it requires stable production techniques and management control. The resulting commercial market is based on experience, rather than a deep understanding of the technology . . . an engineering profession emerges when . . . science becomes sufficiently mature to support purposeful practice and design evolution with predictable outcomes.

1 Kurzweil [Kur05] presents a reasoned technical argument that predicts a strong artificial intelligence (that will pass the Turing Test) by 2029 and suggests that the evolution of humans and machines will begin to merge by 2045. The vast majority of readers of this book will live to see whether this, in fact, comes to pass.

We would argue that the industry has achieved "purposeful practice," but that "predictable outcomes" have remained illusive.

As MobileApps and WebApps begin to dominate the software landscape, Shaw identifies challenges that "emerge from the deep interdependencies between very complex systems and their users" [Sha09]. She argues that the knowledge base that leads to "purposeful practice" has been democratized by the specialized social networks that now populate the Web. For example, rather than referencing a centrally controlled software engineering handbook, a software engineer can pose a problem on an appropriate forum and obtain a crowd-sourced solution that draws from the experience of many other developers. The proposed solution if often critiqued in real time, with alternatives and adaptations offered as options.

But this is not the level of discipline that many demand. As Shaw states: "[P]roblems facing software engineers are increasingly situated in complex social contexts, and delineating the problem's boundaries is increasingly difficult" [Sha09]. As a consequence, isolating the scientific underpinnings of a discipline remains a challenge. At this point in the history of our field, it is reasonable to state that "the discovery of new software engineering ideas is, by now, naturally incremental and evolutionary" [Erd10].

38.3 Observing Software Engineering Trends

Barry Boehm [Boe08] suggests that "software engineers [will] face the often formidable challenges of dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence, but they also have opportunities to make significant contributions that will make a difference for the better." But what are the trends that will enable you to face these challenges in the years ahead?

Quote:

"I think there is a world market for maybe five computers."

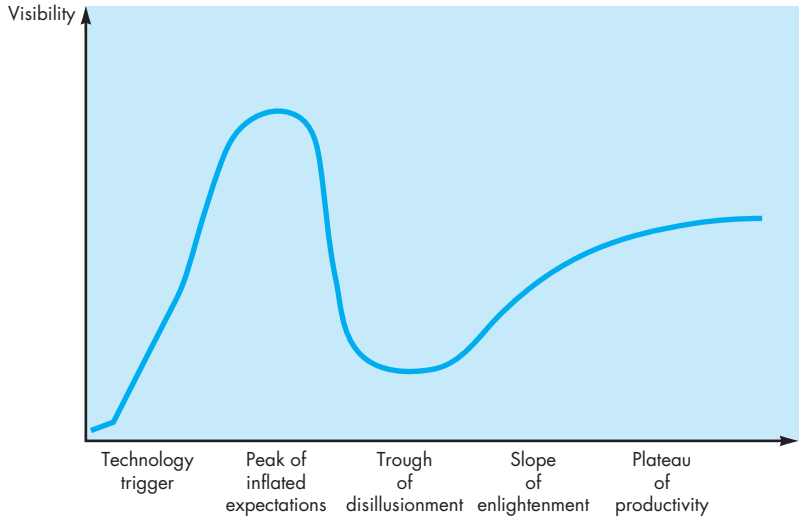
Thomas Watson, chairman of IBM, 1943

In the introduction to this chapter, we noted that "soft trends" have a significant impact on the overall direction of software engineering. But other ("harder") research- and technology-oriented trends remain important. Research trends "are driven by general perceptions of the state of the art and the state of the practice, by researcher perceptions of practitioner needs, by national funding programs that rally around specific strategic goals, and by sheer technical interest" [Mil00b]. Technology trends occur when research trends are extrapolated to meet industry needs and are shaped by market-driven demand.

In Section 38.1, we discussed the S-curve model for technology evolution. The S-curve is appropriate for considering the long-term effects of core technologies as they evolve. But what of more modest, short-term innovations, tools, and methods? The Gartner Group [Gar08]—a consultancy that studies technology trends across many industries—has developed a *hype cycle for emerging technologies,*

FIGURE 38.2

The Gartner Group's hype cycle for emerging technologies [Gar08]



KEY POINT

The "hype cycle" presents a realistic view of short-term technology integration. The long-term trend, however, is exponential.

represented in Figure 38.2. Not every software engineering technology makes it all the way through the hype cycle. In some cases, disillusionment is justified and the technology is relegated to obscurity.

38.4 IDENTIFYING "SOFT TRENDS"

Quote:

"640K ought to be enough for anybody."

Bill Gates, chairman of Microsoft, 1981

Each nation with a substantial IT industry has a set of unique characteristics that define the manner in which business is conducted, the organizational dynamics that arise within a company, the distinct marketing issues that apply to local customers, and the overriding culture that dictates all human interaction. However, some trends in each of these areas are universal and have as much to do with sociology, anthropology, and group psychology (often referred to as the "soft sciences") as they do with academic or industrial research.

Connectivity and collaboration (enabled by high-bandwidth communication) has already led to software teams that do not occupy the same physical space (telecommuting and part-time employment in a local context). One team collaborates with other teams that are separated by time zones, primary language, and culture. Software engineering must respond with an overarching process model for "distributed teams" that is agile enough to meet the demands of immediacy but disciplined enough to coordinate disparate groups.

Globalization leads to a diverse workforce (in terms of language, culture, problem resolution, management philosophy, communication priorities, and person-to-person interaction). This, in turn, demands a flexible organizational structure. Different teams (in different countries) must respond to engineering problems in a way that best accommodates their unique needs, while at the

same time fostering a level of uniformity that allows an overall global project to proceed. This type of organization suggests fewer levels of management and a greater emphasis on team-level decision making. It can lead to greater agility, but only if communication mechanisms have been established so that every team can understand project and technical status (via networked groupware) at any time. Software engineering methods and tools can help achieve some level of uniformity (teams speak the same "language" implemented through specific methods and tools). Software process can provide the framework for the instantiation of these methods and tools.

In some world regions (the United States and Europe are examples), the population is aging. This undeniable demographic (and cultural trend) implies that many experienced software engineers and managers will be leaving the field over the coming decade. The software engineering community must respond with viable mechanisms that capture the knowledge of these aging managers and technologists (e.g., the use of *patterns* (Chapter 16) is a step in the right direction), so that it will be available to future generations of software workers. In other regions of the world, the number of young people available to the software industry is exploding. This provides an opportunity to mold a software engineering culture without the burden of 50 years of "old-school" prejudices.

It is estimated that over 1 billion new consumers will enter the worldwide marketplace over the next decade. Consumer spending in emerging economies will grow to over $12 trillion in 10 years [ATK12]. There is little doubt that a nontrivial percentage of this spending will be applied to products and services that have a digital component—that are software based or software driven. The implication—an increasing demand for new software. The question then is, "Can new software engineering technologies be developed to meet this worldwide demand?" Modern market trends are often driven by the supply side.[2] In other cases, demand-side requirements drive the market. In either case, a cycle of innovation and demand progresses in a way that sometimes makes it difficult to determine which came first!

Finally, human culture itself will impact the direction of software engineering. Every generation establishes its own imprint on local culture, and yours will be no different. Faith Popcorn [Pop08], a well-known consultant who specializes in cultural trends, characterizes them in the following manner: "Our Trends are not fads. Our Trends endure. Our Trends evolve. They represent underlying forces, first causes, basic human needs, attitudes, aspirations. They help us navigate the world, understand what's happening and why, and prepare for what is yet to come." A detailed discussion of how modern cultural trends will have an impact on software engineering is best left to those who specialize in the "soft sciences."

2 Supply side adopts a "build it and they will come" approach to markets. Unique technologies are created, and consumers flock to adopt them–sometimes!

38.4.1 Managing Complexity

When the first edition of this book was written (1982), digital consumer products as we now know them today didn't exist, and mainframe-based systems containing a million lines of source code (LOC) were considered to be quite large. Today, it is not uncommon for small digital devices to encompass between 60,000 to 200,000 lines of custom software, coupled with a few million LOC for operating system features. Modern computer-based systems containing 10 to 50 million lines of code are not uncommon.[3] In the relatively near future, systems[4] requiring over 1 billion LOC will begin to emerge.[5]

Think about that for a moment!

Consider the interfaces for a billion LOC system, both to the outside world, to other interoperable systems, to the Internet (or its successor), and to the millions of internal components that must all work together to make this computing monster operate successfully. Is there a reliable way to ensure that all of these connections will allow information to flow properly?

Consider the project itself. How do we manage the work flow and track progress? Will conventional approaches scale upward by orders of magnitude?

Consider the number of people (and their locations) who will be doing the work, the coordination of people and technology, the unrelenting flow of changes, the likelihood of a multiplatform, multioperating system environment. Is there a way to manage and coordinate people who are working on a monster project?

Consider the engineering challenge. How can we analyze tens of thousands of requirements, constraints, and restrictions in a way that ensures that inconsistency and ambiguity, omissions, and outright errors are uncovered and corrected? How can we create a design architecture that is robust enough to handle a system of this size? How can software engineers establish a change management system that will have to handle hundreds of thousands of changes?

Consider the challenge of quality assurance. How can we perform verification and validation in a meaningful way? How do you test a 1 billion LOC system?

In the early days, software engineers attempted to manage complexity in what can only be described as an ad hoc fashion. Today, we use process, methods, and tools to keep complexity under control. But tomorrow? Is our current approach up to the task?

Quote:

"There is no reason anyone would want a computer in their home."

Ken Olson, President, Chairman and Founder of Digital Equipment Corp., 1977

3 For example, modern PC operating systems (e.g., Linux, Mac OS, and Windows) have between 30 and 60 million LOC. Operating system software for mobile devices can exceed 2 million LOC.

4 In reality, this "system" will actually be a system of systems–hundreds of interoperable applications working together to achieve some overall objective.

5 Not all complex systems are large. A relatively small application (say, less than 100,000 LOC can still be exceedingly complex.

38.4.2 Open-World Software

KEY POINT

Open-world software encompasses ambient intelligence, context-aware apps, and pervasive computing.

Concepts such as ambient intelligence,[6] context-aware applications, and pervasive/ubiquitous computing—all focus on integrating software-based systems into an environment far broader that a PC, a mobile computing device, or any other digital device. These separate visions of the near-term future of computing collectively suggest "open-world software"—software that is designed to adapt to a continually changing environment "by self-organizing its structure and self-adapting its behavior" [Bar06b].

To help illustrate the challenges that software engineers will face in the foreseeable future, consider the notion of *ambient intelligence* (amI). Ducatel [Duc01] defines amI in the following way: "People are surrounded by intelligent, intuitive interfaces that are embedded in all kinds of objects. The ambient intelligence environment is capable of recognizing and responding to the presence of different individuals [while working] in a seamless, unobtrusive way."

With the widespread use of low-cost, yet increasingly powerful smartphones, we are well on our way to ubiquitous amI systems. The challenge for software engineers is to develop apps that provide ever-increasing functionality in products of all types—functionality that adapts to user needs while at the same time protecting privacy and providing security.

38.4.3 Emergent Requirements

At the beginning of a software project, there's a truism that applies equally to every stakeholder involved: "You don't know what you don't know." That means that customers rarely define "stable" requirements. It also means that software engineers cannot always foresee where ambiguities and inconsistencies lie. Requirements change—but that's nothing new.

ADVICE

Because emergent requirements are already a reality, your organization should consider adopting an incremental process model.

As systems become more complex, it follows that even a rudimentary attempt to state comprehensive requirements is doomed to failure. A statement of overall goals may be possible, delineation of intermediate objectives can be accomplished, but stable requirements—not a chance! Requirements will emerge as everyone involved in the engineering and construction of a complex system learns more about it, the environment in which it is to reside, and the users who will interact with it.

This reality implies a number of software engineering trends. First, process models must be designed to embrace change and adopt the basic tenets of the agile philosophy (Chapter 5). Next, methods that yield engineering models (e.g., requirements and design models) must be used judiciously because those models will change repeatedly as more knowledge about the system is acquired.

6 A worthwhile and quite detailed introduction to *ambient intelligence* can be found at **www.emergingcommunication.com/volume6.html**. More information can be obtained at **www.ambientintelligence.org/.**

Finally, tools that support both process and methods must make adaptation and change easy.

But there is another aspect to emergent requirements. The vast majority of software developed to date assumes that the boundary between the software-based system and its external environment is stable. The boundary may change, but it will do so in a controlled manner, allowing the software to be adapted as part of a regular software maintenance cycle. This assumption is beginning to change. Open-world software (Section 38.4.2) demands that software "adapt and react to changes dynamically, even if they're unanticipated" [Bar06b].

By their nature, emergent requirements lead to change. How do we control the evolution of a widely used application or system over its lifetime, and what effect does this have on the way we design software?

As the number of changes grows, the likelihood of unintended side effects also grows. This should be a cause for concern as complex systems with emergent requirements become the norm. The software engineering community must develop methods that help software teams predict the impact of change across an entire system, thereby mitigating unintended side effects. Today, our ability to accomplish this is severely limited.

38.4.4 The Talent Mix

As software-based systems become more complex, as communication and collaboration among global teams becomes commonplace, as emergent requirements (with the resultant flow of changes) become the norm, the very nature of a software engineering team may change. Each software team must bring a variety of creative talent and technical skills to its part of a complex system, and the overall process must allow the output of these islands of talent to merge effectively.

Alexandra Weber-Morales [Mor05] suggests the talent mix of a "software dream team." The *Brain* is a chief architect who is able to navigate the demands of stakeholders and map them into a technology framework that is both extensible and implementable. The *Data Grrl* is a database and data structures guru who "blasts through rows and columns with profound understanding of predicate logic and set theory as it pertains to the relational model." The *Blocker* is a technical leader (manager) who allows the team to work free of interference from other teams while at the same time ensuring that collaboration is occurring. The *Hacker* is a consummate programmer who is at home with patterns and languages and can use both effectively. The *Gatherer* "deftly discovers system requirements with . . . anthropological insight" and accurately expresses them with clarity.

38.4.5 Software Building Blocks

All of us who have fostered a software engineering philosophy have emphasized the need for reuse—of source code, object-oriented classes, components, patterns, and COTS software. Although the software engineering

community has made progress as it attempts to capture past knowledge and reuse proven solutions, a significant percentage of the software that is built today continues to be built "from scratch." Part of the reason for this is a continuing desire (by stakeholders and software engineering practitioners) for "unique solutions."

In the hardware world, original equipment manufacturers (OEMs) of digital devices use application-specific standard products (ASSPs) produced by silicon vendors almost exclusively. This "merchant hardware" provides the building blocks necessary to implement everything from a smartphone to a wearable computing device. Increasingly, the same OEMs are using "merchant software"—software building blocks designed specifically for a unique application domain [e.g., VoIP devices]. Michael Ward [War07] comments:

> One advantage of the use of software components is that the OEM can leverage the functionality provided by the software without having to develop in-house expertise in the specific functions or invest developer time on the effort to implement and validate the components. Other advantages include the ability to acquire and deploy only the specific set of functionalities that are needed for the system, as well as the ability to integrate these components into an already-existing architecture.

In addition to components packaged as merchant software, there is an increasing tendency to adopt *software platform solutions* that "incorporate collections of related functionalities, typically provided within an integrated software framework" [War07]. A software platform frees an OEM from the work associated with developing base functionality and instead allows the OEM to dedicate software effort on those features that differentiate its product.

38.4.6 Changing Perceptions of "Value"

During the last quarter of the 20th century, the operative question that businesspeople asked when discussing software was: Why does it cost so much? That question is rarely asked today and has been replaced by: Why can't we get it (software and/or the software-based product) sooner?

When computer software is considered, the modern perception of value is changing from business value (cost and profitability) to customer values that include: speed of delivery, richness of functionality, and overall product quality.

38.4.7 Open Source

Who owns the software you or your organization uses? Increasingly, the answer is "everyone." The "open source" movement has been described in the following manner [OSO12]: "Open source is a development method for software that harnesses the power of distributed peer review and transparency of process. The promise of open source is better quality, higher reliability, more

Quote:

"But what is it good for?"

Engineer at the Advanced Computing Systems Division of IBM, 1968, commenting on the microchip

flexibility, lower cost, and an end to predatory vendor lock-in." The term *open source,* when applied to computer software, implies that software engineering work products (models, source code, test suites) are open to the public and can be reviewed and extended (with controls) by anyone with interest and permission.

If you have further interest, Weber [Web05] provides a worthwhile introduction, Feller and his colleagues [Fel07] have edited a comprehensive and objective anthology that considers the benefits and problems associated with open source, and Brown [Bro12] provides a more technical discussion.

38.5 Technology Directions

Quote:

"The proper artistic response to digital technology is to embrace it as a new window on everything that's eternally human, and to use it with passion, wisdom, fearlessness and joy."

Ralph Lombreglia

We always seem to think that software engineering will change more rapidly than it does. A new "hyped" technology (it could be a new process, a unique method, or an exciting tool) is introduced, and pundits suggest that "everything" will change. But software engineering is about far more than technology—it's about people and their ability to communicate their needs and innovate to make those needs a reality. Whenever people are involved, change occurs slowly in fits and starts. It's only when a "tipping point" [Gla02] is reached that a technology cascades across the software engineering community and broad-based change truly does occur.

In this section we'll examine a few trends in process, methods, and tools that are likely to have some influence on software engineering over the next decade. Will they lead to a tipping point? We'll just have to wait and see.

38.5.1 Process Trends

It can be argued that all of the business, organizational, and cultural trends discussed in Section 38.4 reinforce the need for process. But do the frameworks discussed in Chapter 37 provide a road map into the future? Will process frameworks evolve to find a better balance between discipline and creativity? Will the software process adapt to the differing needs of stakeholders who procure software, those who build it, and those who use it? Can it provide a means for reducing risk for all three constituencies at the same time?

These and many other questions remain open. In the paragraphs that follow, we have adapted six ideas proposed by Conradi and Fuggetta [Con02] to suggest possible process trends.

? What process trends are likely over the next decade?

1. *As SPI frameworks evolve, they will emphasize "strategies that focus on goal orientation and product innovation"* [Con02]. In the fast-paced world of software development, long-term SPI strategies rarely survive in a dynamic business environment. Too much changes too quickly. This means that a stable, step-by-step road map for SPI may have to be replaced with

a framework that emphasizes short-term goals that have a product orientation.

2. *Because software engineers have a good sense of where the process is weak, process changes should generally be driven by their needs and should start form the bottom up.* Conradi and Fuggetta [Con02] suggest that future SPI activities should "use a simple and focused scorecard to start with, not a large assessment." By focusing SPI efforts narrowly and working from the bottom up, practitioners will begin to see substantive changes early—changes that make a real difference in the way that software engineering work is conducted.

3. *Automated software process technology (SPT) will move away from global process management (broad-based support of the entire software process) to focus on those aspects of the software process that can best benefit from automation.* No one is against tools and automation, but in many instances, SPT has not met its promise (see Section 38.3). To be most effective, it should focus on umbrella activities (Chapter 3)—the most stable elements of the software process.

4. *Greater emphasis will be placed on the return on investment of SPI activities.* In Chapter 37, you learned that return on investment (ROI) can be defined as:

$$\text{ROI} = \frac{\Sigma(\textit{benefits}) - \Sigma(\textit{costs})}{\Sigma(\textit{costs})} \times 100\%$$

To date, software organizations have struggled to clearly delineate "benefits" in a quantitative manner. It can be argued [Con02] that "we therefore need a standardized market-value model . . . to account for software improvement initiatives."

5. *As time passes, the software community may come to understand that expertise in sociology and anthropology may have as much or more to do with successful SPI as other, more technical disciplines.* Above all else, SPI changes organizational culture, and cultural change involves individuals and groups of people. Conradi and Fuggetta [Con02] correctly note that "software developers are knowledge workers. They tend to respond negatively to top-level dictates on how to do work or change processes." Much can be learned by examining the sociology of groups to better understand effective ways to introduce change.

6. *New modes of learning may facilitate the transition to a more effective software process.* In this context, "learning" implies learning from successes and mistakes. A software organization that collects metrics (Chapters 30 and 32) allows itself to understand how elements of a process affect the quality of the end product.

38.5.2 The Grand Challenge

There is one trend that is undeniable—software-based systems will undoubtedly become bigger and more complex as time passes. It is the engineering of these large, complex systems, regardless of delivery platform or application domain, that poses the "grand challenge" [Bro06] for software engineers. Manfred Broy [Bro06] suggests that software engineers can meet "the daunting challenge of complex software systems development" by creating new approaches to understanding system models and using those models as a basis for the construction of high-quality next-generation software.

As the software engineering community develops new model-driven approaches (discussed briefly later in this section) to the representation of system requirements and design, the following characteristics [Bro06] must be addressed:

? What system characteristics must analysts and designers consider for future apps?

- *Multifunctionality*—As digital devices evolve, they have begun to deliver a rich set of sometimes unrelated functions. The mobile phone, once considered a straightforward communication device, has become a powerful pocket computer that performs a wide spectrum of functions that are arguably more important than making a phone call. As Broy [Bro06] notes, "[E]ngineers must describe the detailed context in which the functions will be delivered and, most important, must identify the potentially harmful interactions between the system's different features."
- *Reactivity and timeliness*—Digital devices increasingly interact with the real world and must react to external stimuli in a timely manner. They must interface with a broad array of sensors and must respond in a time frame that is appropriate to the task at hand. New methods must be developed that (1) help software engineers predict the timing of various reactive features and (2) implement those features in a way that makes the feature less machine dependent and more portable.
- *New modes of user interaction*—Open-world trends for software mean that new modes of interaction must be modeled and implemented. Whether these new approaches use multitouch interfaces, voice recognition, or direct mind interfaces,[7] new generations of software for digital devices must accommodate them.
- *Complex architectures*—A luxury automobile has over 2,000 functions controlled by software residing within a complex hardware architecture that includes multiple processors, a sophisticated bus structure, actuators, sensors, an increasingly sophisticated human interface, and many

7 A brief discussion of direct mind interfaces can be found at **http://en.wikipedia.org/wiki/Brain-computer_interface**, and a commercial example which continues to evolve is described at **http://au.gamespot.com/news/6166959.html**

safety-rated components. Even more complex systems are on the immediate horizon, presenting significant challenges for software designers.

- *Heterogeneous, distributed systems*—The real-time components of any modern embedded system can be connected via an internal bus, a wireless network, or across the Internet (or all three).
- *Criticality*—Software has become the pivotal component in virtually all business-critical systems and in most safety-critical systems. Yet, the software engineering community has only begun to apply even the most basic principles of software safety.
- *Maintenance variability*—The life of software within a digital device rarely lasts beyond 3 to 5 years, but the complex avionics systems within an aircraft has a useful life of at least 20 years. Automobile software falls somewhere in between. Should this have an impact on design?

Broy [Bro06] argues that these and other software characteristics can be managed only if the software engineering community develops a more effective distributed and collaborative software engineering philosophy, better requirements engineering approaches, a more robust approach to model-driven development, and better software tools. In the sections that follow we'll explore each of these areas briefly.

38.5.3 Collaborative Development

KEY POINT

Collaboration involves the timely dissemination of information and an effective process for communication and decision making.

It seems too obvious to state, but we'll do so anyway: *software engineering is an information technology*. From the onset of any software project, every stakeholder must share information—about basic business goals and objectives, about specific system requirements, about architectural design issues, about almost every aspect of the software to be built.

Today, software engineers collaborate across time zones and international boundaries. Every one of them must share information. The same holds for open-source projects in which hundreds or thousands of software developers work to build an open-source app. Again, information must be disseminated so that open collaboration can occur.

38.5.4 Requirements Engineering

Basic requirements engineering actions—elicitation, elaboration, negotiation, specification, and validation—were presented in Chapters 8 through 11. The success or failure of these actions has a very strong influence on the success or failure of the entire software engineering process. And yet, requirements engineering (RE) has been compared to "trying to put a hose clamp around jello" [Gon04]. As we've noted in many places throughout this book, software requirements have a tendency to keep changing, and with the advent of open-world systems, emergent requirements (and near-continuous change) may become the norm.

Today, most "informal" requirements engineering approaches begin with the creation of user scenarios (e.g., use cases). More formal approaches create one or more requirements models and use these as a basis for design. Formal methods enable a software engineer to represent requirements using a verifiable mathematical notation. All can work reasonably well when requirements are stable, but do not readily solve the problem of dynamic or emergent requirements.

There are a number of distinct requirements engineering research directions including natural language processing from translated textual descriptions into more structured representations (e.g., analysis classes), greater reliance on databases for structuring and understanding software requirements, the use of RE patterns to describe typical problems and solutions when requirements engineering tasks are conducted, and goal-oriented requirements engineering. However, at the industry level, RE actions remain relatively informal and surprisingly basic. To improve the manner in which requirements are defined, the software engineering community will likely implement three distinct subprocesses as RE is conducted [Gli07]: (1) improved knowledge acquisition and knowledge sharing that allows more complete understanding of application domain constraints and stakeholder needs, (2) greater emphasis on iteration as requirements are defined, and (3) more effective communication and coordination tools that enable all stakeholders to collaborate effectively.

The RE subprocesses noted in the preceding paragraph will only succeed if they are properly integrated into an evolving approach to software engineering. As pattern-based problem solving and component-based solutions begin to dominate many application domains, RE must accommodate the desire for agility (rapid incremental delivery) and the inherent emergent requirements that result. The notion of a static "software specification" is beginning to disappear, to be replaced by "value-driven requirements" [Som05] derived as stakeholders respond to features and functions delivered in early software increments.

38.5.5 Model-Driven Software Development

KEY POINT

Model-driven approaches address a continuing challenge for all software developers—how to represent software at a higher level of abstraction than code.

Software engineers grapple with abstraction at virtually every step in the software engineering process. As design commences, architectural and component-level abstractions are represented and assessed. They must then be translated into a programming language representation that transforms the design (a relatively high level of abstraction) into an operable system with a specific computing environment (a low level of abstraction). *Model-driven software development*[8] couples domain-specific modeling languages with transformation engines and generators in a way that facilitates the representation of abstraction at high levels and then transforms it into lower levels [Sch06].

8 The term *model-driven engineering* (MDE) is also used.

Domain-specific modeling languages (DSMLs) represent "application structure, behavior and requirements within particular application domains" and are described with meta-models that "define the relationships among concepts in the domain and precisely specify the key semantics and constraints associated with these domain concepts" [Sch06]. The key difference between a DSML and a general-purpose modeling language such as UML (Appendix 1) is that the DSML is tuned to design concepts inherent in the application domain and can therefore represent relationships and constraints among design elements in an efficient manner.

38.5.6 Postmodern Design

In an interesting article on software design in the "postmodern era," Philippe Kruchten [Kru05] makes the following observation:

> Computer science hasn't achieved the grand narrative that explains it all, the *big picture*—we haven't found the fundamental laws of software that would play the role that the fundamental laws of physics play in other engineering disciplines. We still live with the bitter aftertaste of the Internet bubble burst and the Y2K doomsday. So, in this postmodern era, where it seems that everything matters a bit yet not much really matters, what are the next directions for software design?

Part of any attempt to understand trends in software design is to establish boundaries for design. Where does requirements engineering stop and design begin? Where does design stop and code generation begin? The answers to these questions are not as easy as they might first appear. Even though the requirements model should focus on "what," not "how," every analyst does a bit of design and almost all designers do a bit of analysis. Similarly, as the design of software components moves closer to algorithmic detail, a designer begins to represent the component at a level of abstraction that is close to code.

38.5.7 Test-Driven Development

Requirements drive design, and design establishes a foundation for construction. This simple software engineering reality works reasonably well and is essential as a software architecture is created. However, a subtle change can provide significant benefit when component-level design and construction are considered.

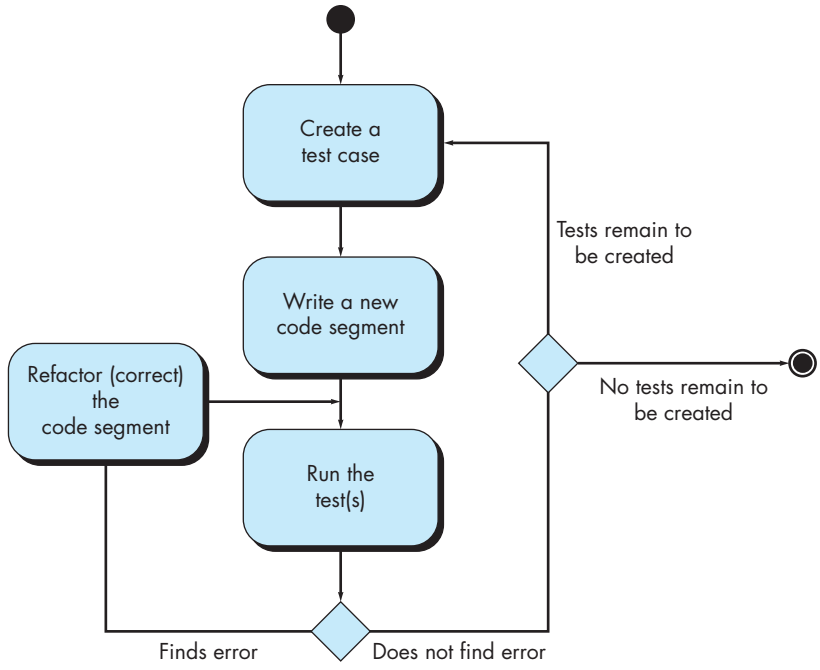
In *test-driven development* (TDD), requirements for a software component serve as the basis for the creation of a series of test cases that exercise the interface and attempt to find errors in the data structures and functionality delivered by the component. TDD is not really a new technology but rather a trend that emphasizes the design of test cases *before* the creation of source code.[9]

---

9 Recall that Extreme Programming (Chapter 5) emphasizes this approach as part of its agile process model.

FIGURE 38.3

Test-driven development process flow



The TDD process follows the simple procedural flow illustrated in Figure 38.3. Before the first small segment of code is created, a software engineer creates a test to exercise the code (to try to make the code fail). The code is then written to satisfy the test. If it passes, a new test is created for the next segment of code to be developed. The process continues until the component is fully coded and all tests execute without error. However, if any test succeeds in finding an error, the existing code is refactored (corrected) and all tests created to that point are re-executed. This iterative flow continues until there are no tests left to be created, implying that the component meets all requirements defined for it.

During TDD, code is developed in very small increments (one subfunction at a time), and no code is written until a test exists to exercise it. You should note that each iteration results in one or more new tests that are added to a regression test suite that is run with every change. This is done to ensure that the new code has not generated side effects that cause errors in the older code. If you have further interest in TDD, see [Bec04b], [Ste10], or [Whi12].

38.6 TOOLS-RELATED TRENDS

Hundreds of industry-grade software engineering tools are introduced each year. The majority are provided by tools vendors who claim that their tool will improve project management, or requirements analysis, or design modeling, or

code generation, or testing, or change management, or any of the many software engineering activities, actions, and tasks discussed throughout this book. Other tools have been developed as open-source offerings. The majority of open-source tools focus on "programming" activities with a specific emphasis on the construction activity (particularly code generation). Still other tools grow out of research efforts at universities and government labs. Although they have appeal in very limited applications, the majority are not ready for broad industry application.

At the industry level, the most comprehensive tools packages form *software engineering environments* (SEE)[10] that integrate a collection of individual tools around a central database (repository). When considered as a whole, an SEE integrates information across the software process and assists in the collaboration that is required for many large, complex software-based systems. But current environments are not easily extensible (it's difficult to integrate a COTS tool that is not part of the package) and tend to be general purpose (i.e., they are not application domain specific). There is also a substantial time lag between the introduction of new technology solutions (e.g., model-driven software development) and the availability of viable SEEs that support the new technology.

Future trends in software tools will follow two distinct paths—a *human-focused path* that responds to some of the "soft trends" discussed in Section 38.4, and a technology-centered path that addresses new technologies (Section 38.5) as they are introduced and adopted.

The soft trends discussed in Section 38.4—the need to manage complexity, accommodate emergent requirements, establish process models that embrace change, coordinate global teams with a changing talent mix, among others—suggest a new era in which tools support for stakeholder collaboration will become as important as tools support for technology.

Agility in software engineering (Chapter 5) is achieved when stakeholders work as a team. Therefore, the trend toward collaborative SEEs will provide benefits even when software is developed locally. But what of the technology tools that complement the system and components that empower better collaboration?

One of the dominant trends in technology tools is the creation of a tool set that supports model-driven development (Section 38.5.5) with an emphasis on architecture-driven design. Oren Novotny [Nov04] suggests that the model rather than the source code becomes the central software engineering focus:

> Platform independent models are created in UML and then undergo various levels of transformation to eventually wind up as source code for a specific platform. It stands to reason then, that the model, not the file, should become the new unit of output. A model has many different views at different levels of abstraction. At the highest level, platform independent components can be specified in analysis; at the lowest level there is a platform specific implementation that reduces to a set of classes in code.

10 The term *integrated development environment* (IDE) is also used.

Novotny argues that a new generation of tools will work in conjunction with a repository to create models at all necessary levels of abstraction, establish relationships between the various models, translate models at one level of abstraction to another level (e.g., translate a design model into source code), manage changes and versions, and coordinate quality control and assurance actions against the software models.

In addition to complete software engineering environments, point-solution tools that address everything from requirements gathering to design/code refactoring to testing will continue to evolve and become more functionally capable. In some instances, modeling and testing tools targeted at a specific application domain will provide enhanced benefit when compared to their generic equivalents.

38.7 Summary

The trends that have an effect on software engineering technology often come from the business, organizational, market, and cultural arenas. These "soft trends" can guide the direction of research and the technology that is derived as a consequence of research.

As a new technology is introduced, it moves through a life cycle that does not always lead to widespread adoption, even though original expectations are high. The degree to which any software engineering technology gains widespread adoption is tied to its ability to address the problems posed by both soft and hard trends.

Soft trends—the growing need for connectivity and collaboration, global projects, knowledge transfer, the impact of emerging economies, and the influence of human culture itself—lead to a set of challenges that spans managing complexity and emergent requirements to juggling an ever-changing talent mix among geographically dispersed software teams.

Hard trends—the ever-accelerating pace of technology change—flow out of soft trends and affect the structure of the software and scope of the process and the manner in which a process framework is characterized. Collaborative development, new forms of requirements engineering, model-based and test-driven development, and postmodern design will change the methods landscape. Tools environments will respond to a growing need for communication and collaboration and at the same time integrate domain-specific point solutions that may change the nature of current software engineering tasks.

Problems and Points to Ponder

38.1. Get a copy of the best-selling book *The Tipping Point* by Malcolm Gladwell (available via Google Book Search), and discuss how his theories apply to the adoption of new software engineering technologies.

38.2. Why does open-world software present a challenge to conventional software engineering approaches?

38.3. Review the Gartner Group's *hype cycle for emerging technologies*. Select a well-known technology product and present a brief history that illustrates how it traveled along the curve. Select another well-known technology product that did not follow the path suggested by the hype curve.

38.4. What is a "soft trend"?

38.5. You're faced with an extremely complex problem that will require a lengthy solution. How would you go about addressing the complexity and crafting a solution?

38.6. What are "emergent requirements" and why do they present a challenge to software engineers?

38.7. Select an open-source development effort (other than Linux), and present a brief history of its evolution and relative success.

38.8. Describe how you think the software process will change over the next decade.

38.9. You're based in Los Angeles and are working on a global software engineering team. You and colleagues in London, Mumbai, Hong Kong, and Sydney must edit a 245-page requirements specification for a large system. The first editing pass must be completed in three days. Describe the ideal online tool set that would enable you to collaborate effectively.

38.10. Describe model-driven software development in your own words. Do the same for test-driven development.

Further Readings and Information Sources

Books that discuss the road ahead for software and computing span a vast array of technical, scientific, economic, political, and social issues. Kurweil (*The Singularity Is Near,* Penguin Books, 2005) and (*How to Create a Mind,* Viking, 2012) presents a compelling look at a world that will change in truly profound ways by the middle of this century. Sterling (*Tomorrow Now,* Random House, 2002) reminds us that real progress is rarely orderly and efficient. Books by Nanz (*The Future of Software Engineering,* Springer, 2010) and Draheim and his colleagues (*Software Engineering Tools: Trends of Software Engineering Tools and Platforms, 2010*) discuss trends in software development. Meisel (*The Software Society: Cultural and Economic Impact,* Trafford, 2013), Saylor (*The Mobile Wave: How Mobile Intelligence Will Change Everything,* Vanguard Press, 2012), Dourish and Bell (*Divining a Digital Future: Mess and Mythology in Ubiquitous Computing,* MIT Press, 2011) and Teich (*Technology and the Future,* 12th ed., Wadworth, 2012) present thoughtful essays on the societal impact of technology and how changing culture shapes technology. Philips and Naisbitt (*High Tech/High Touch,* Nicholas Brealey, 2001) note that many of us have become "intoxicated" with high technology and that the "great irony of the high-tech age is that we've become enslaved to devices that were supposed to give us freedom." Zey (*The Future Factor,* Transaction Publishers, 2004) discusses forces that will shape human destiny during this century. Negroponte's *(Being Digital,* Alfred A. Knopf, 1995) was a best seller in the mid-1990s and continues to provide an insightful view of computing and its overall impact.

As software becomes part of the fabric of virtually every facet of our lives, "cyberethics" has evolved as an important topic of discussion. Books by Quninn (*Ethics for the Information Age,* 5th ed., Addison-Wesley, 2012), Spinello (*Cyberethics: Morality and Law in Cyberspace*, 4th ed., Jones & Bartlett Publishers, 2010), Tavini (*Ethics and Technology,* 3rd ed., Wiley, 2010), Halbert and Ingulli (*Cyberethics,* 2nd ed., South-Western College Publishers, 2004), and Baird and his colleagues (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consider the topic in detail. The U.S. government has published

a voluminous report on CD-ROM (*21st Century Guide to Cybercrime,* Progressive Management, 2003) that considers all aspects of computer crime, intellectual property issues, and the National Infrastructure Protection Center (NIPC).

A wide variety of information sources on future directions in software-related technologies and software engineering are available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman.**

CHAPTER

39 Concluding Comments

Key Concepts

artificial intelligence 863
communication ... 862
ethics 865
future 864
information spectrum 864
knowledge 863
people 861
responsibility 865
software revisited 861

In the 38 chapters that have preceded this one, we've explored a process for software engineering that encompasses management procedures and technical methods, basic concepts and principles, specialized techniques, people-oriented activities and tasks that are amenable to automation, paper-and-pencil notation, and software tools. We have argued that measurement, discipline, and an overriding focus on agility and quality will result in software that meets the customer's needs, software that is reliable, software that is maintainable, software that is better. Yet, we have never promised that software engineering is a panacea.

Software and systems technologies remain a challenge for every software professional and every company that builds computer-based systems. Although he wrote these words with a 20th-century outlook, Max Hopper [Hop90] accurately describes the current state of affairs:

> Because changes in information technology are becoming so rapid and unforgiving, and the consequences of falling behind are so irreversible, companies will either master the technology or die . . . Think of it as a technology treadmill. Companies will have to run harder and harder just to stay in place.

Changes in software engineering technology are indeed "rapid and unforgiving," but at the same time real progress is often quite slow. By the time a decision is made to adopt a new process, method, or tool; conduct the training necessary to understand its application; and introduce the technology into the software development culture, something newer (and even better) has come along, and the process begins anew.

Quick Look

What is it? As we come to the end of a relatively long journey through software engineering, it's time to put things into perspective and make a few concluding comments.

Who does it? Authors like us. When you come to the end of a long and challenging book, it's nice to wrap things up in a meaningful way.

Why is it important? It's always worthwhile to remember where we've been and to consider where we're going.

What are the steps? We'll consider where we've been and address some of the core issues and some directions for the future.

What is the work product? A discussion that will help you understand the big picture.

How do I ensure that I've done it right? That's difficult to accomplish in real time. It's only after a number of years that any of us can tell whether the software engineering concepts, principles, methods, and techniques discussed in this book have helped you to become a better software engineer.

One thing we've learned over our years in this field is that software engineering practitioners are "fashion conscious." The road ahead will be littered with the carcasses of exciting new technologies (the latest fashion) that never really made it (despite the hype). It will be shaped by more modest technologies that somehow modify the direction and width of the thoroughfare. We discussed a few of those in Chapter 38.

In this concluding chapter we'll take a broader view and consider where we've been and where we're going from a more philosophical perspective.

39.1 The Importance of Software—Revisited

The importance of computer software can be stated in many ways. In Chapter 1, software was characterized as a differentiator. The function delivered by software differentiates products, systems, and services and provides competitive advantage in the marketplace. But software is more than a differentiator. When taken as a whole, software engineering work products generate the most important commodity that any individual, business, or government can acquire—information.

In Chapter 38, we briefly discussed open-world computing—a technology that is fundamentally changing our perception of computers, the things that we do with them (and they do for us), and our perception of information as a guide, a commodity, and a necessity. We also noted that software required to support open-world computing will present dramatic new challenges for software engineers. But far more important, the growing pervasiveness of computer software will present even more dramatic challenges for society as a whole. Whenever a technology has a broad impact—an impact that can save lives or endanger them, build businesses or destroy them, inform government leaders or mislead them—it must be "handled with care."

39.2 People and the Way They Build Systems

Quote:

"Future shock [is] the shattering stress and disorientation that we induce in individuals by subjecting them to too much change in too short a period of time."

Alvin Toffler

The software required for high-technology systems becomes more and more complex with each passing year, and the size of resultant programs increases proportionally. The rapid growth in the size of the "average" program would present us with few problems if it wasn't for one simple fact: As program size increases, the number of people who must work on the program must also increase.

Experience indicates that as the number of people on a software project team increases, the overall productivity of the group may suffer. One way around this problem is to create a number of software engineering teams, thereby compartmentalizing people into individual working groups. However, as the number of software engineering teams grows, communication between them becomes as difficult and time consuming as communication between individuals. Worse, communication (between individuals or teams) tends to be inefficient—that is,

too much time is spent transferring too little information content, and all too often, important information "falls into the cracks."

If the software engineering community is to deal effectively with the communication dilemma, the road ahead for software engineers must include radical changes in the way individuals and teams communicate with one another. In Chapter 38, we discussed collaborative environments that may provide dramatic improvements in the ways teams communicate.

In the final analysis, communication is the transfer of knowledge, and the acquisition (and transfer) of knowledge is changing in profound ways. As search engines become increasingly sophisticated, social networking and crowd-sourcing morph into a development tools, and Web 2.0 applications provide better synergy, the speed and quality of knowledge transfer will grow exponentially.

If past history is any indication, it is fair to say that people themselves will not change. However, the ways in which they communicate, the environment in which they work, the manner in which they acquire knowledge, the methods and tools that they use, the discipline that they apply, and therefore, the overall culture for software development will change in significant and even profound ways.

SafeHome

Conclusion?

The scene: Doug Miller's office.

The players: Doug Miller (manager of the *SafeHome* software engineering group) and Vinod Raman (a member of the product software engineering team).

The conversation:

Doug: I'm really pleased that we got it done without too much drama.

Vinod (sighing and leaning back in his chair): Yeah, but the project grew, didn't it.

Doug: And you're surprised? When we started *SafeHome*, marketing thought a desktop app would do the trick, and then . . .

Vinod (smiling): And then, the Web and mobility took over.

Doug: But we all learned a lot.

Vinod: We did. The tech stuff was interesting, but the software engineering stuff is probably what allowed us to get it done close to schedule.

Doug: Yeah, that and hard work by all of you guys. What are you seeing from customer support? How's quality in the field?

Vinod: There are a few issues, but nothing really serious. We're on it. In fact, I gotta meet with Jamie on one of them in five minutes.

Doug: Before you go . . .

Vinod (on his way out the door): I know, more work, right?

Doug: Engineering has developed a new sensor . . . very high tech . . . we'll need to integrate it in *SafeHome II.*

Vinod: *SafeHome II?*

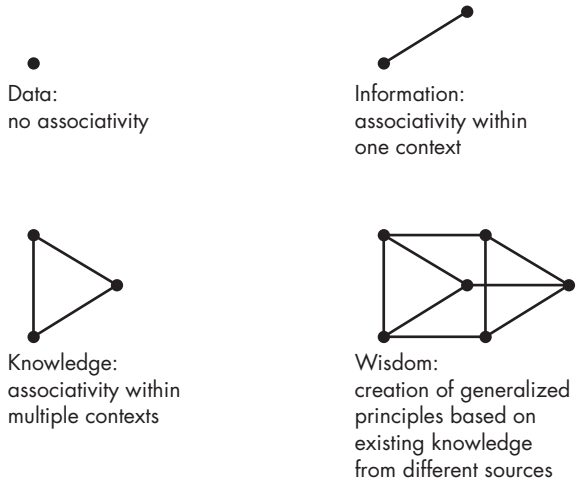
Doug: Yeah, *SafeHome II.* We'll begin planning next week.

39.3 New Modes for Representing Information

Over the history of computing, a subtle transition has occurred in the terminology that is used to describe software development work performed for the business community. Fifty years ago, the term *data processing* was the operative

FIGURE 39.1

An "information" spectrum



Quote:

"The best preparation for good work tomorrow is to do good work today."

Elbert Hubbard

phrase for describing the use of computers in a business context. Today, data processing has given way to another phrase—*information technology*—that implies the same thing but presents a subtle shift in focus. The emphasis is not merely to process large quantities of data but rather to extract meaningful information from this data. Obviously, this was always the intent, but the shift in terminology reflects a far more important shift in management philosophy.

When software applications are discussed today, the words *data, information,* and *content* occur repeatedly. We encounter the word *knowledge* in some *artificial intelligence* applications, but its use is relatively rare. Virtually no one discusses *wisdom* in the context of software applications.

Data is raw information—collections of facts that must be processed to be meaningful. Information is derived by associating facts within a given context. Knowledge associates information obtained in one context with other information obtained in a different context. Finally, wisdom occurs when generalized principles are derived from disparate knowledge. Each of these four views of "information" is represented schematically in Figure 39.1.

To date, the vast majority of all software has been built to process data or information. Software engineers are now equally concerned with systems that process knowledge.[1] Knowledge is two-dimensional. Information collected on a variety of related and unrelated topics is connected to form a body of fact that we call *knowledge.* The key is our ability to associate information from a variety of different sources that may not have any obvious connection and combine it in a way that provides us with some distinct benefit.[2]

1 The rapid growth of data mining and data warehousing technologies reflect this growing trend.

2 The semantic Web (Web 2.0) allows the creation of "mashups" that may provide a facile mechanism for achieving this.

Quote:

"Wisdom is the power that enables us to use knowledge for the benefit of ourselves and others."

Thomas J. Watson

To illustrate the progression from data to knowledge, consider census data indicating that the birthrate in 1996 in the United States was 4.9 million. This number represents a data value. Relating this piece of data with birthrates for the preceding 40 years, we can derive a useful piece of information—aging baby boomers of the 1950s and early 1960s made a last-gasp effort to have children prior to the end of their child-bearing years. In addition, gen-Xers began their childbearing years. The census data can then be connected to other seemingly unrelated pieces of information. For example, the current number of elementary school teachers who will retire during the next decade, the number of college students graduating with degrees in primary and secondary education, the pressure on politicians to hold down taxes and therefore limit pay increases for teachers.

All of these pieces of information can be combined to formulate a representation of knowledge—there will be significant pressure on the education system in the United States in the early 21st century, and this pressure will continue for a number of decades. Using this knowledge, a business opportunity may emerge. There may be significant opportunity to develop new modes of learning that are more effective and less costly than current approaches.

The road ahead for software leads to systems that process knowledge. We have been processing data using computers for more than 50 years and extracting information for more than three decades. One of the most significant challenges facing the software engineering community is to build systems that take the next step along the spectrum—systems that extract knowledge from data and information in a way that is practical and beneficial.

39.4 The Long View

In Section 39.3, we suggested that the road ahead leads to systems that "process knowledge." But the future of computing in general and software-based systems in particular may lead to events that are considerably more profound.

In a fascinating book that is must reading for every person involved in computing technologies, Ray Kurzweil [Kur05] suggests that we have reached a time when "the pace of technological change will be so rapid, its impact so deep, that human life will be irreversibly transformed." Kurzweil[3] makes a compelling argument that we are currently at the "knee" of an exponential growth curve that will lead to enormous increases in computing capacity over the next two

3 It's important to note that Kurzweil is not a run-of-the mill science fiction writer or a futurist without portfolio. He is a serious technologist who (from Wikipedia) "has been a pioneer in the fields of optical character recognition (OCR), text-to-speech synthesis, speech recognition technology, and electronic keyboard instruments."

decades. When coupled with equivalent advances in nanotechnology, genetics, and robotics, we may approach a time in the middle part of this century when the distinction between humans (as we know them today) and machines begins to blur—a time when human evolution accelerates in ways that are both frightening (to some) and spectacular (to others).

Quote:

"You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future."

Steve Jobs

By sometime in the coming decade, Kurzweil argues that computing capacity and the requisite software will be sufficient to model every aspect of the human brain [Kur13] —all of the physical connections, analog processes, and chemical overlays. When this occurs, human beings will take the first step toward achieving "strong AI (artificial intelligence)," and as a consequence, machines that truly do think (using today's conventional use of the word). But there will be a fundamental difference. Human brain processes are exceedingly complex and only loosely connected to external informal sources. They are also computationally slow, even in comparison to today's computing technology. When full human brain emulation occurs, "thought" will occur at speeds thousands of times more rapid than its human counterpart with intimate connections to a sea of information (think of the present-day Web as a primitive example). The result is . . . well . . . so fantastical that it's best left to Kurzweil to describe.

It's important to note that not everyone believes that the future Kurzweil describes is a good thing. In a now famous essay entitled "The Future Doesn't Need Us," Bill Joy [Joy00], one of the founders of Sun Microsystems, argues that "robotics, genetic engineering, and nanotech are threatening to make humans an endangered species." His arguments predicting a technology dystopia represent a counterpoint to Kurzweil's predicted utopian future. Both should be seriously considered as software engineers take one of the lead roles in defining the long view for the human race.

39.5 The Software Engineer's Responsibility

Software engineering has evolved into a respected, worldwide profession. As professionals, software engineers should abide by a code of ethics that guides the work that they do and the products that they produce. An ACM/IEEE-CS Joint Task Force has produced a *Software Engineering Code of Ethics and Professional Practices* (Version 5.1). The code [ACM12] states:

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
>
> 1. PUBLIC—Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.
8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

WebRef

A complete discussion of the ACM/IEEE code of ethics can be found at **seeri.etsu.edu/Codes/default.shtm.**

Although each of these eight principles is equally important, an overriding theme appears: a software engineer should work in the public interest. On a personal level, a software engineer should abide by the following rules:

- Never steal data for personal gain.
- Never distribute or sell proprietary information obtained as part of your work on a software project.
- Never maliciously destroy or modify another person's programs, files, or data.
- Never violate the privacy of an individual, a group, or an organization.
- Never hack into a system for sport or profit.
- Never create or promulgate a computer virus or worm.
- Never use computing technology to facilitate discrimination or harassment.

Over the past decade, certain members of the software industry have lobbied for protective legislation that [SEE03]: (1) allows companies to release software without disclosing known defects, (2) exempts developers from liability for any damages resulting from these known defects, (3) constrains others from disclosing defects without permission from the original developer, (4) allows the incorporation of "self-help" software within a product that can disable (via remote command) the operation of the product, and (5) exempts developers of software with "self-help" from damages should the software be disabled by a third party.

Like all legislation, debate often centers on issues that are political, not technological. However, many people (including us) feel that protective legislation, if improperly drafted, conflicts with the software engineering code of ethics by indirectly exempting software engineers from their responsibility to produce high-quality software.

39.6 A Final Comment from RSP

It has been almost three and a half decades since work on the first edition of this book began. I [RSP] can still recall sitting at my desk as a young professor, writing the manuscript for a book on a subject that few people cared about and even fewer really understood. I remember the rejection letters from publishers, who argued (politely, but firmly) that there would never be a market for a book on "software engineering." Luckily, McGraw-Hill decided to give it a try,[4] and the rest, as they say, is history.

Since the first edition, this book has changed dramatically—in scope, in size, in style, and in content. Like software engineering, it has grown and (I hope) matured over the years.

An engineering approach to the development of computer software is now conventional wisdom. Debate continues on the "right paradigm," the importance of agility, the degree of automation, and the most effective methods. But the underlying principles of software engineering are now accepted throughout the industry. Why, then, have we seen their broad adoption only recently?

The answer, I think, lies in the difficulty of technology transition and the cultural change that accompanies it. Even though most of us appreciate the need for an engineering discipline for software, we struggle against the inertia of past practice and face new application domains (and the developers who work in them) that appear ready to repeat the mistakes of the past. To ease the transition we need many things—an agile, adaptable, and sensible software process; more effective methods; more powerful tools; better acceptance by practitioners and support from managers; and no small dose of education.

You may not agree with every approach described in this book. Some of the techniques and opinions are controversial; others must be tuned to work well in different software development environments. It is my sincere hope, however, that *Software Engineering: A Practitioner's Approach* has delineated the problems we face, demonstrated the strength of software engineering concepts, and provided a framework of methods and tools.

4 Actually, credit should go to Peter Freeman and Eric Munson, who convinced McGraw-Hill that it was worth a shot. Almost 2 million copies later, it's fair to say they made a good decision.

As we move further into the 21st century, software continues to be the most important product and the most important industry on the world stage. Its impact and importance have come a long, long way. And yet, a new generation of software developers must meet many of the same challenges that faced earlier generations. Let us hope that the people who meet the challenge—software engineers—will have the wisdom to develop systems that improve the human condition.

APPENDIX

1

An Introduction to UML[1]

Key Concepts

activity diagram . . 881
class diagram 870
communication diagram 880
dependency. 872
deployment diagram 874
generalization. . . . 871
interaction frames 878
multiplicity 872
object constraint language 887
sequence diagram . 876
state diagram 884
stereotype 871
swimlanes 883
use-case diagram . 875

The *Unified Modeling Language* (UML) is "a standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system" [Boo05]. In other words, just as building architects create blueprints to be used by a construction company, software architects create UML diagrams to help software developers build the software. If you understand the vocabulary of UML (the diagrams' pictorial elements and their meanings), you can much more easily understand and specify a system and explain the design of that system to others.

Grady Booch, Jim Rumbaugh, and Ivar Jacobson developed UML in the mid-1990s with much feedback from the software development community. UML merged a number of competing modeling notations that were in use by the software industry at the time. In 1997, UML 1.0 was submitted to the Object Management Group, a nonprofit consortium involved in maintaining specifications for use by the computer industry. UML 1.0 was revised to UML 1.1 and adopted later that year. The current standard is UML 2.3[2] and is now an ISO standard. Because this standard is new, many older references, such as [Gam95] do not use UML notation.

UML 2.3 provides 13 different diagrams for use in software modeling. In this appendix, we will discuss only *class, deployment, use case, sequence, communication, activity,* and *state* diagrams. These diagrams are used in this edition of *Software Engineering: A Practitioner's Approach.*

You should note that there are many optional features in UML diagrams. The UML language provides these (sometimes arcane) options so that you can express all the important aspects of a system. At the same time, you have the flexibility to suppress those parts of the diagram that are not relevant to the aspect being modeled in order to avoid cluttering the diagram with irrelevant details. Therefore, the omission of a particular feature does not mean that the feature is absent; it may mean that the feature was suppressed. In this appendix, we will not present exhaustive coverage of all the features of the UML diagrams. Instead, we focus on the standard options, especially those options that have been used in this book.

1 This appendix has been contributed by Dale Skrien and has been adapted from his book, *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). All content is used with permission.

2 **http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/** and **http://www.omg.org/spec/UML/2.3/Superstructure/PDF/**. These two complementary specifications constitute a complete specification for the UML 2 modeling language

Class Diagrams

To model classes, including their attributes, operations, and their relationships and associations with other classes,[3] UML provides a *class diagram*. A class diagram provides a static or structural view of the system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram.

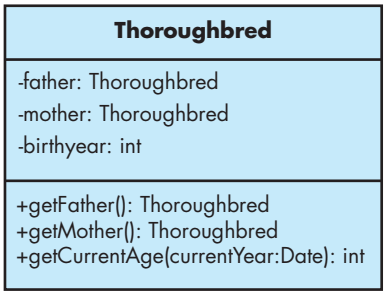
The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. Attributes can be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed. For example, an object might always know the current time and be able to return it to you whenever you ask, in which case it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An *operation* refers to what objects of the class can do. It is usually implemented as a *method* of the class.

Figure A1.1 presents a simple example of a **Thoroughbred** class that models thoroughbred horses. It has three attributes displayed—**mother**, **father**, and **birth year**. The diagram also shows three operations: *getCurrentAge()*, *getFather()*, and *getMother()*. There may be other suppressed attributes and operations not shown in the diagram.

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding –, #, ~, or +, indicating,

Figure A1.1

A class diagram for a Thoroughbred class



3 If you are unfamiliar with object-oriented concepts, a brief introduction is presented in Appendix 2.

respectively, *private, protected, package,* or *public* visibility. In Figure A1.1, all attributes have private visibility, as indicated by the leading minus sign (−). You can also specify that an attribute is a static or class attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type.

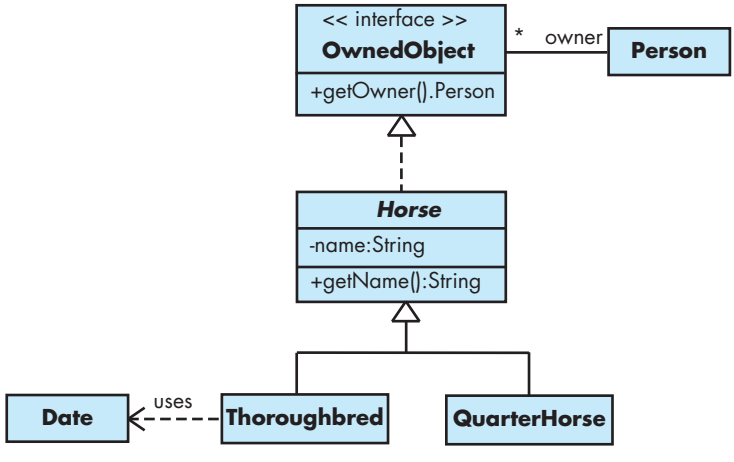
An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the **Horse** class in Figure A1.2 for an example. An interface is indicated by adding the phrase "«interface»" (called a *stereotype*) above the name. See the **OwnedObject** interface in Figure A1.2. An interface can also be represented graphically by a hollow circle.

It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This section is particularly useful when transitioning from CRC cards (Chapter 10) to class diagrams in that the responsibilities listed on the CRC cards can be added to this fourth section in the class box in the UML diagram before creating the attributes and operations that carry out these responsibilities. This fourth section is not shown in any of the figures in this appendix.

Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a *generalization*. For example, in Figure A1.2, the **Thoroughbred** and **QuarterHorse** classes are shown to be subclasses of the **Horse** abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a *realization*. For example, in Figure A1.2, the **Horse** class implements or realizes the **OwnedObject** interface.

FIGURE A1.2

A class diagram regarding horses



An *association* between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A1.2, there is an association between **OwnedObject** and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class.

An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the first class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A1.2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A1.2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type.

A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them. However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A1.2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses."

The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means

that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a **Person** class and a **SocialSecurityNumber** class in a class diagram. A multiplicity specified by "1..*" means one or more, and a multiplicity specified by "0..*" or just " * " means zero or more. An * was used as the multiplicity on the **OwnedObject** end of the association with class **Person** in Figure A1.2 because a **Person** can own zero or more objects.

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

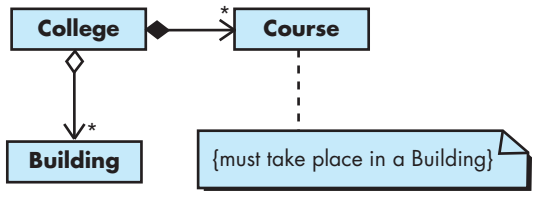
An *aggregation* is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered a "part" of the class at the diamond end of the association. A *composition* is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner. See Figure A1.3 for examples of aggregation and composition.

A **College** has an aggregation of **Building** objects, which represent the buildings making up the campus. The college also has a collection of courses. If the college were to fold, the buildings would still exist (assuming the college wasn't physically destroyed) and could be used for other things, but a **Course** object has no use outside of the college at which it is being offered. If the college were to cease to exist as a business entity, the **Course** object would no longer be useful and so it would also cease to exist.

Another common element of a class diagram is a *note*, which is represented by a box with a dog-eared corner and is connected to other icons by a dashed line. It can have arbitrary content (text and graphics) and is similar to a programming language comment. It might contain information about the role of a class or constraints that all objects of that class must satisfy. If the contents are a constraint, braces surround the contents. Note the constraint attached to the **Course** class in Figure A1.3.

FIGURE A1.3

The relationship between Colleges, Courses, and Buildings



Deployment Diagrams

A UML *deployment diagram* focuses on the structure of the software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments. Suppose, for example, you are developing a Web-based graphics-rendering package. Users of your package will use their Web browser to go to your website and enter rendering information. Your website would render a graphical image according to the user's specification and send it back to the user. Because graphics rendering can be computationally expensive, you decide to move the rendering itself off the Web server and onto a separate platform. Therefore, there will be three hardware devices involved in your system: the Web client (the users' computer running a browser), the computer hosting the Web server, and the computer hosting the rendering engine.

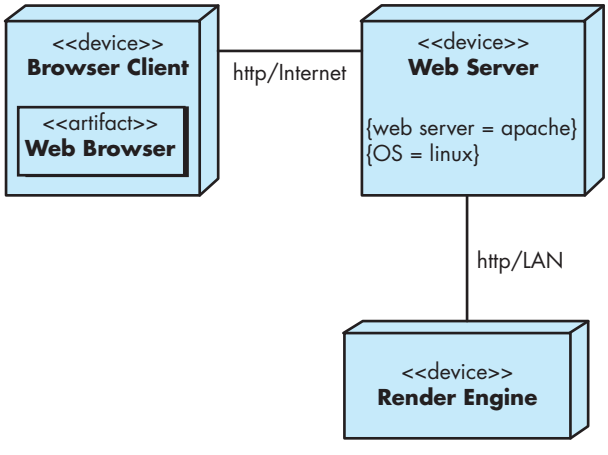
Figure A1.4 shows the deployment diagram for such a package. In such a diagram, hardware components are drawn in boxes labeled with "«device»". Communication paths between hardware components are drawn with lines with optional labels. In Figure A1.4, the paths are labeled with the communication protocol and the type of network used to connect the devices.

Each node in a deployment diagram can also be annotated with details about the device. For example, in Figure A1.4, the browser client is depicted to show that it contains an artifact consisting of the Web browser software. An artifact is typically a file containing software running on a device. You can also specify tagged values, as is shown in Figure A1.4 in the Web server node. These values define the vendor of the Web server and the operating system used by the server.

Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label "«execution environment»". These nodes represent systems, such as operating systems, that can host other software programs.

Figure A1.4

A deployment diagram



Use-Case Diagrams

Use cases (Chapters 8 and 9) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we'll create some for a software application for managing an online digital music store. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD). Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

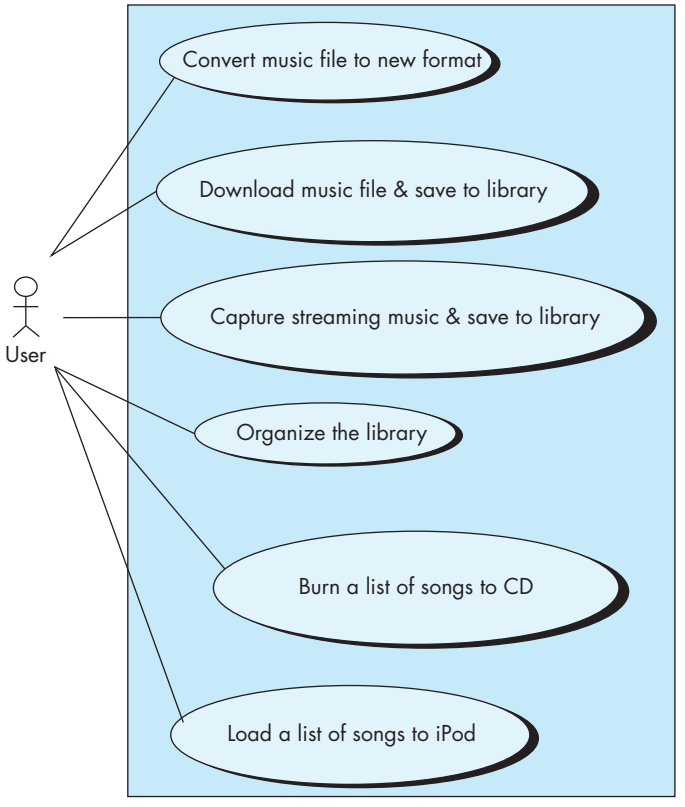
In this diagram, the stick figure represents an *actor* (Chapter 8) that is associated with one category of user (or other interaction element). Complex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the actors are outside the system.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod or smartphone. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps.

FIGURE A1.5

A use-case diagram for the music system



Such inclusion is indicated in use-case diagrams, as in Figure A1.6, by means of a dashed arrow labeled «include» connecting a use case with an included use case.

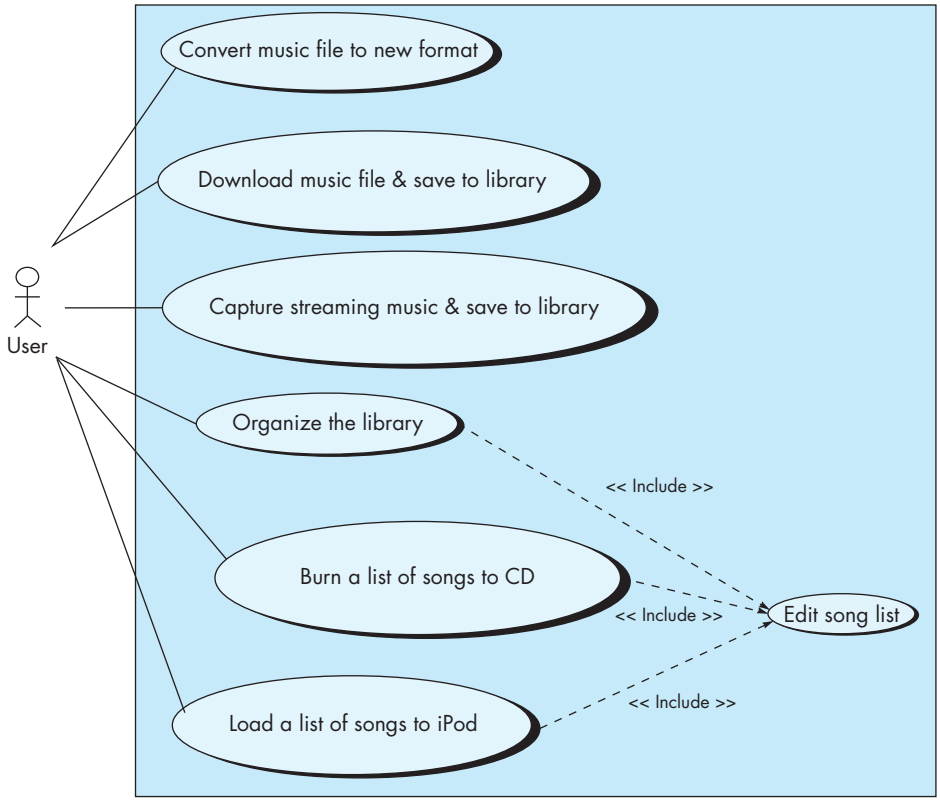
A use-case diagram, because it displays all use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In our digital music organizer, we would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most valuable contribution of use cases to the software development process is the textual description of each use case, not the overall use-case diagram [Fow04]. It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

Sequence Diagrams

In contrast to class diagrams and deployment diagrams, which show the static structure of a software component, a *sequence diagram* is used to show the dynamic communications between objects during execution of a task. It shows the temporal order in which messages are sent between the objects to accomplish

FIGURE A1.6

A use-case diagram with included use cases



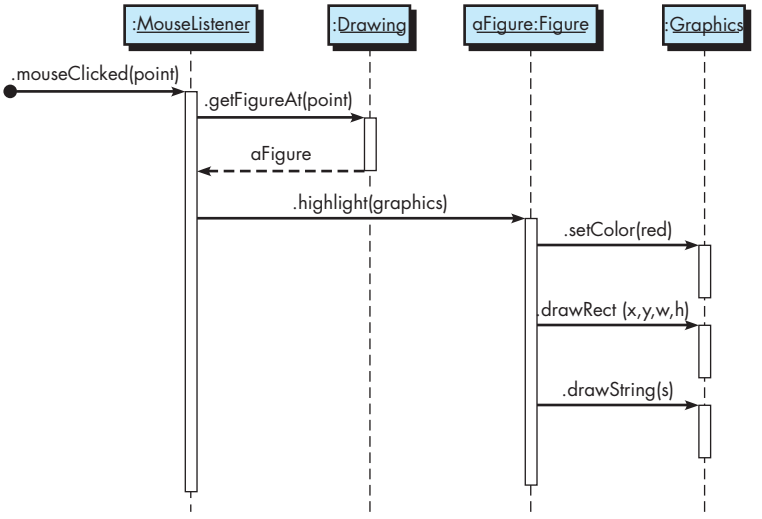
that task. One might use a sequence diagram to show the interactions in one use case or in one scenario of the software system.

In Figure A1.7, you see a sequence diagram for a drawing program. The diagram shows the steps involved in highlighting a figure in the drawing when it has been clicked. Each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes. If the box represents an object (as is the case in all our examples), then inside the box you can optionally state the type of the object preceded by the colon. You can also precede the colon and type by a name for the object, as shown in the third box in Figure A1.7. Below each box there is a dashed line called the *lifeline* of the object. The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.

A sequence diagram shows method calls using horizontal arrows from the *caller* to the *callee*, labeled with the method name and optionally including its parameters, their types, and the return type. For example, in Figure A1.7, the **MouseListener** calls the **Drawing**'s *getFigureAt()* method. When an object is executing a method (that is, when it has an activation frame on the stack), you can

FIGURE A1.7

A sample sequence diagram



optionally display a white bar, called an *activation bar*, down the object's lifeline. In Figure A1.7, activation bars are drawn for all method calls. The diagram can also optionally show the return from a method call with a dashed arrow and an optional label. In Figure A1.7, the *getFigureAt()* method call's return is shown labeled with the name of the object that was returned. A common practice, as we have done in Figure A1.7, is to leave off the return arrow when a void method has been called, since it clutters up the diagram while providing little information of importance. A black circle with an arrow coming from it indicates a *found message* whose source is unknown or irrelevant.

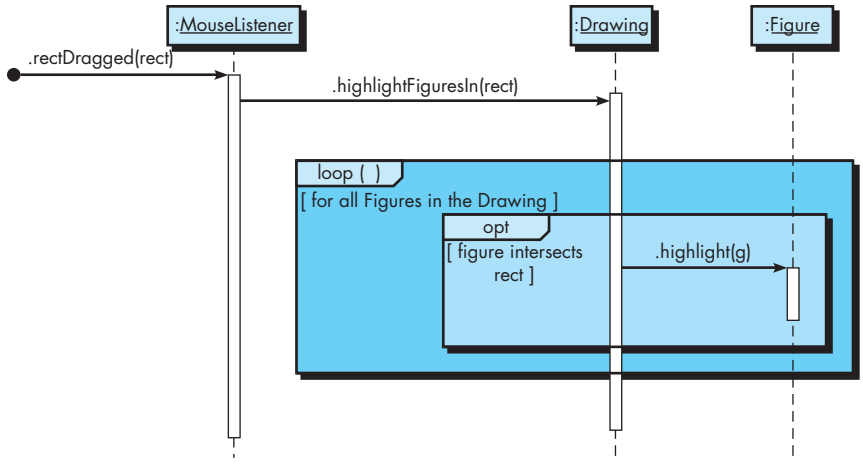
You should now be able to understand the task that Figure A1.7 is displaying. An unknown source calls the *mouseClicked()* method of a **MouseListener**, passing in the point where the click occurred as the argument. The **MouseListener** in turn calls the *getFigureAt()* method of a **Drawing**, which returns a **Figure.** The **MouseListener** then calls the highlight method of **Figure**, passing in a **Graphics** object as an argument. In response, **Figure** calls three methods of the **Graphics** object to draw the figure in red.

The diagram in Figure A1.7 is very straightforward and contains no conditionals or loops. If logical control structures are required, it is probably best to draw a separate sequence diagram for each case. That is, if the message flow can take two different paths depending on a condition, then draw two separate sequence diagrams, one for each possibility.

If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use *interaction frames*, which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent. Figure A1.8 illustrates this, showing the process involved

FIGURE A1.8

A sequence diagram with two interaction frames



in highlighting all figures inside a given rectangle. The **MouseListener** is sent the **rectDragged** message. The **MouseListener** then tells the drawing to highlight all figures in the rectangle by called the method *highlightFigures()*, passing the rectangle as the argument. The method loops through all **Figure** objects in the **Drawing** object and, if the **Figure** intersects the rectangle, the **Figure** is asked to highlight itself. The phrases in square brackets are called *guards*, which are Boolean conditions that must be true if the action inside the interaction frame is to continue.

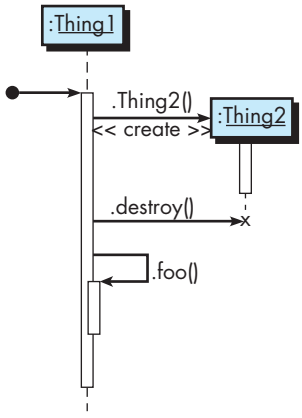
There are many other special features that can be included in a sequence diagram. For example:

1. You can distinguish between synchronous and asynchronous messages. Synchronous messages are shown with solid arrowheads while asynchronous messages are shown with stick arrowheads.
2. You can show an object sending itself a message with an arrow going out from the object, turning downward, and then pointing back to the same object.
3. You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label) to an object's box. In this case, the box will appear lower in the diagram than the boxes corresponding to objects already in existence when the action begins.
4. You can show object destruction by a big X at the end of the object's lifeline. Other objects can destroy an object, in which case an arrow points from the other object to the X. An X is also useful for indicating that an object is no longer usable and so is ready for garbage collection.

The last three features are all shown in the sequence diagram in Figure A1.9.

FIGURE A1.9

Creation, destruction, and loops in sequence diagrams



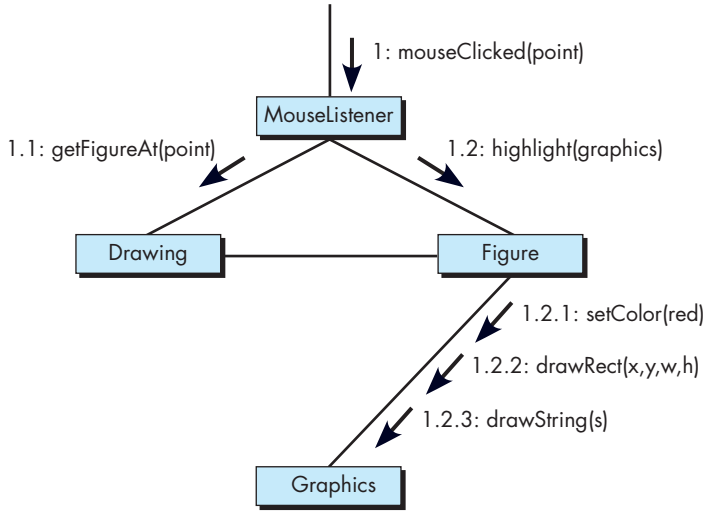
COMMUNICATION DIAGRAMS

The UML *communication diagram* (known as a "collaboration diagram" in UML 1.X) provides another indication of the temporal order of the communications but emphasizes the relationships among the objects and classes instead of the temporal order. A communication diagram is illustrated in Figure A1.10, which displays the same actions shown in the sequence diagram in Figure A1.7.

In a communication diagram the interacting objects are represented by rectangles. Associations between objects are represented by lines connecting the rectangles. There is typically an incoming arrow to one object in the diagram that starts the sequence of message passing. That arrow is labeled with a number and a message name. If the incoming message is labeled with the number 1 and if it causes the receiving object to invoke other messages on other objects, then

FIGURE A1.10

A UML communication diagram



those messages are represented by arrows from the sender to the receiver along an association line and are given numbers 1.1, 1.2, and so forth, in the order they are called. If those messages in turn invoke other messages, another decimal point and number are added to the number labeling these messages, to indicate further nesting of the message passing.

In Figure A1.10, you see that the **mouseClicked** message invokes the methods *getFigureAt()* and then *highlight()*. The *highlight()* message invokes three other messages: *setColor(), drawRect(),* and *drawstring()*. The numbering in each label shows the nesting as well as the sequential nature of each message.

There are many optional features that can be added to the arrow labels. For example, you can precede the number with a letter. An incoming arrow could be labeled **A1: mouseClicked(point).** indicating an execution thread, A. If other messages are executed in other threads, their label would be preceded by a different letter. For example, if the *mouseClicked()* method is executed in thread A but it creates a new thread B and invokes *highlight()* in that thread, then the arrow from **MouseListener** to **Figure** would be labeled **1.B2: highlight(graphics)**.

If you are interested in showing the relationships among the objects in addition to the messages being sent between them, the communication diagram is probably a better option than the sequence diagram. If you are more interested in the temporal order of the message passing, then a sequence diagram is probably better.

Activity Diagrams

A UML *activity diagram* depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.

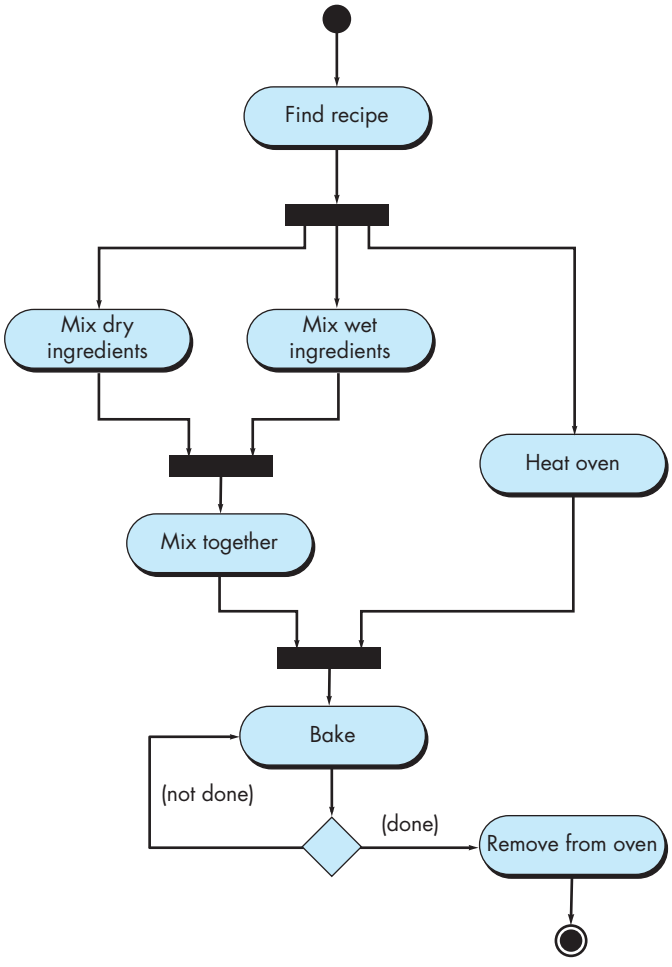
The main component of an activity diagram is an *action* node, represented by a rounded rectangle, which corresponds to a task performed by the software system. Arrows from one action node to another indicate the flow of control. That is, an arrow between two action nodes means that after the first action is complete the second action begins. A solid black dot forms the *initial node* that indicates the starting point of the activity. A black dot surrounded by a black circle is the *final node* indicating the end of the activity.

A *fork* represents the separation of activities into two or more concurrent activities. It is drawn as a horizontal black bar with one arrow pointing to it and two or more arrows pointing out from it. Each outgoing arrow represents a flow of control that can be executed concurrently with the flows corresponding to the other outgoing arrows. These concurrent activities can be performed on a computer using different threads or even using different computers.

Figure A1.11 shows a sample activity diagram involving baking a cake. The first step is finding the recipe. Once the recipe has been found, the dry ingredients and wet ingredients can be measured and mixed and the oven can be

FIGURE A1.11

A UML activity diagram showing how to bake a cake



preheated. The mixing of the dry ingredients can be done in parallel with the mixing of the wet ingredients and the preheating of the oven.

A *join* is a way of synchronizing concurrent flows of control. It is represented by a horizontal black bar with two or more incoming arrows and one outgoing arrow. The flow of control represented by the outgoing arrow cannot begin execution until all flows represented by incoming arrows have been completed. In Figure A1.11, we have a join before the action of mixing together the wet and dry ingredients. This join indicates that all dry ingredients must be mixed and all wet ingredients must be mixed before the two mixtures can be combined. The second join in the figure indicates that, before the baking of the cake can begin, all ingredients must be mixed together, and the oven must be at the right temperature.

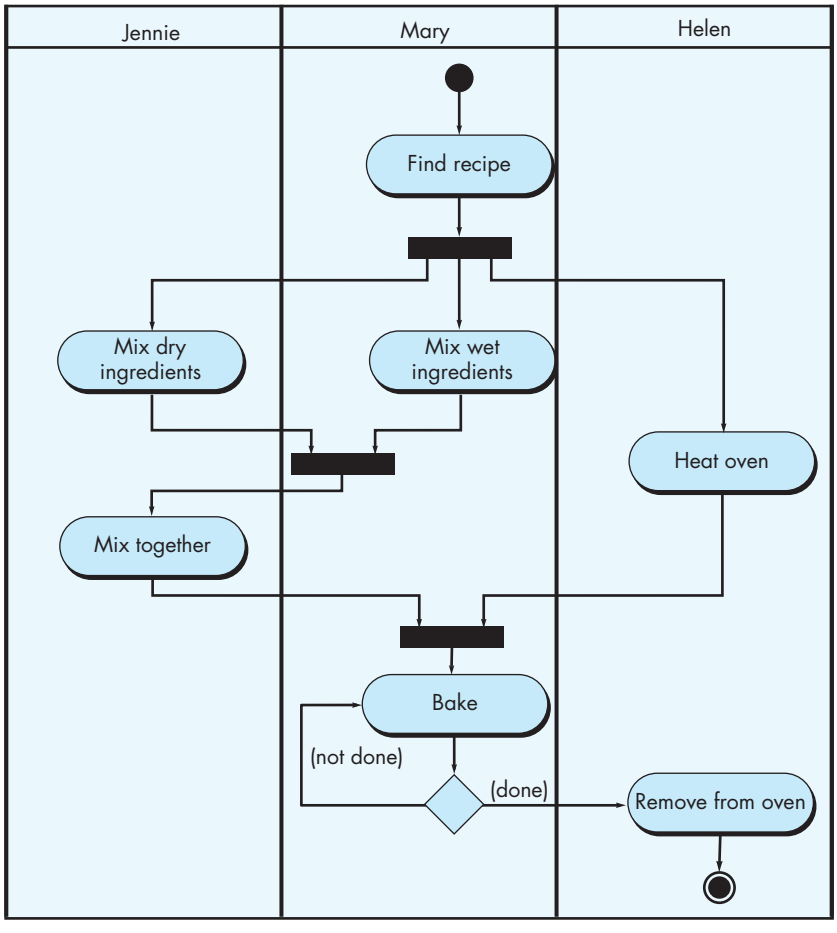
A *decision* node corresponds to a branch in the flow of control based on a condition. Such a node is displayed as a white triangle with an incoming arrow

and two or more outgoing arrows. Each outgoing arrow is labeled with a guard (a condition inside square brackets). The flow of control follows the outgoing arrow whose guard is true. It is advisable to make sure that the conditions cover all possibilities so that exactly one of them is true every time a decision node is reached. Figure A1.11 shows a decision node following the baking of the cake. If the cake is done, then it is removed from the oven. Otherwise, it is baked for a while longer.

One of the things the activity diagram in Figure A1.11 does not tell you is who or what does each of the actions. Often, the exact division of labor does not matter. But if you do want to indicate how the actions are divided among the participants, you can decorate the activity diagram with swimlanes, as shown in Figure A1.12. *Swimlanes*, as the name implies, are formed by dividing the diagram into strips or "lanes," each of which corresponds to one of the participants. All actions in one lane are done by the corresponding participant. In Figure A1.12, Jennie is responsible for mixing the dry ingredients and then mixing

FIGURE A1.12

The cake-baking activity diagram with swimlanes added



the dry and wet ingredients together, Helen is responsible for heating the oven and taking the cake out, and Mary is responsible for everything else.

State Diagrams

The behavior of an object at a particular point in time often depends on the state of the object, that is, the values of its variables at that time. As a trivial example, consider an object with a Boolean instance variable. When asked to perform an operation, the object might do one thing if that variable is *true* and do something else if it is *false*.

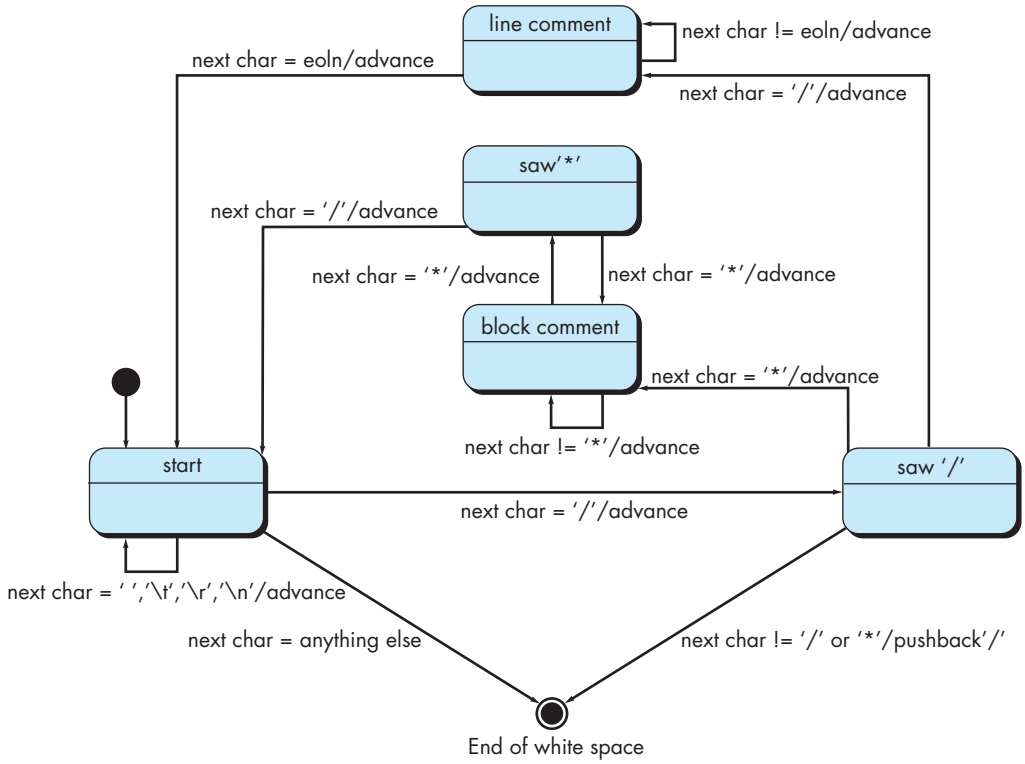
A UML *state diagram* models an object's states, the actions that are performed depending on those states, and the transitions between the states of the object.

As an example, consider the state diagram for a part of a Java compiler. The input to the compiler is a text file, which can be thought of as a long string of characters. The compiler reads characters one at a time and from them determines the structure of the program. One small part of this process of reading the characters involves ignoring "white-space" characters (e.g., the *space, tab, newline,* and *return* characters) and characters inside a comment.

Suppose that the compiler delegates to a **WhiteSpaceAndCommentEliminator** the job of advancing over white-space characters and characters in comments. That is, this object's job is to read input characters until all white-space and comment characters have been read, at which point it returns control to the compiler to read and process non-white-space and noncomment characters. Think about how the **WhiteSpaceAndCommentEliminator** object reads in characters and determines whether the next character is white space or part of a comment. The object can check for white space by testing the next character against " ", "\t", "\n", and "\r". But how does it determine whether the next character is part of a comment? For example, when it sees a "/" for the first time, it doesn't yet know whether that character represents a division operator, part of the /= operator, or the beginning of a line or block comment. To make this determination, **WhiteSpaceAndCommentEliminator** needs to make a note of the fact that it saw a "/" and then move on to the next character. If the character following the "/" is another "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that it is now reading a comment and can advance to the end of the comment without processing or saving any characters. If the character following the first "/" is anything other than a "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that the "/" represents the division operator or part of the /= operator and so it stops advancing over characters.

In summary, as **WhiteSpaceAndCommentEliminator** reads in characters, it needs to keep track of several things, including whether the current character is white space, whether the previous character it read was a "/", whether it is currently reading characters in a comment, whether it has reached the end of

FIGURE A1.13 A state diagram for advancing past white space and comments in Java



comment, and so forth. These all correspond to different states of the **WhiteSpaceAndCommentEliminator** object. In each of these states, **WhiteSpaceAndCommentEliminator** behaves differently with regard to the next character read in.

To help you visualize all the states of this object and how it changes state, you can use a UML state diagram as shown in Figure A1.13. A state diagram displays states using rounded rectangles, each of which has a name in its upper half. There is also a black circle called the "initial pseudostate," which isn't really a state and instead just points to the initial state. In Figure A1.13, the **start** state is the initial state. Arrows from one state to another state indicate transitions or changes in the state of the object. Each transition is labeled with a trigger event, a slash (/), and an activity. All parts of the transition labels are optional in state diagrams. If the object is in one state and the trigger event for one of its transitions occurs, then that transition's activity is performed and the object takes on the new state indicated by the transition. For example, in Figure A1.13, if the **WhiteSpaceAndCommentEliminator** object is in the **start** state and the next character is "/", then **WhiteSpaceAndCommentEliminator** advances past that character and changes to the **saw '/'** state. If the character after the "/" is another "/", then the object advances to the **line comment** state and stays there until it

reads an end-of-line character. If instead the next character after the "/'"is a "*", then the object advances to the **block comment** state and stays there until it sees another "*" followed by a "/", which indicates the end of the block comment. Study the diagram to make sure you understand it. Note that, after advancing past white space or a comment, **WhiteSpaceAndCommentEliminator** goes back to the **start** state and starts over. That behavior is necessary since there might be several successive comments or white-space characters before any other characters in the Java source code.

An object may transition to a final state, indicated by a black circle with a white circle around it, which indicates there are no more transitions. In Figure A1.13, the **WhiteSpaceAndCommentEliminator** object is finished when the next character is not white space or part of a comment. Note that all transitions except the two transitions leading to the final state have activities consisting of advancing to the next character. The two transitions to the final state do not advance over the next character because the next character is part of a word or symbol of interest to the compiler. Note that if the object is in the **saw '/'** state but the next character is not "/" or "*", then the "/" is a division operator or part of the /= operator and so we don't want to advance. In fact, we want to back up one character to make the "/" into the next character, so that the "/" can be used by the compiler. In Figure A1.13, this activity of backing up is labeled as pushback '/'.

A state diagram will help you to uncover missed or unexpected situations. That is, with a state diagram, it is relatively easy to ensure that all possible trigger events for all possible states have been accounted for. For example, in Figure A1.13, you can easily verify that every state has included transitions for all possible characters.

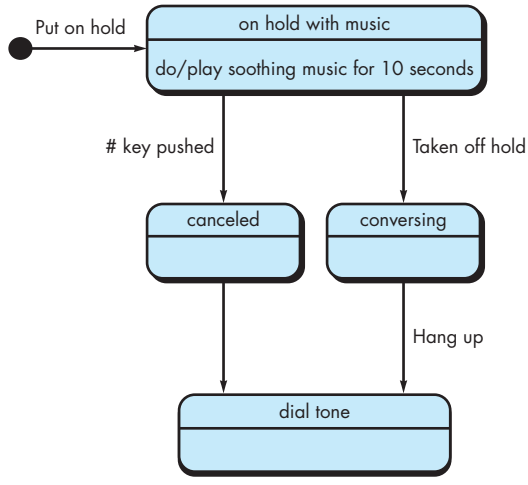
UML state diagrams can contain many other features not included in Figure A1.13. For example, when an object is in a state, it usually does nothing but sit and wait for a trigger event to occur. However, there is a special kind of state, called an *activity state*, in which the object performs some activity, called a *do-activity*, while it is in that state. To indicate that a state is an activity state in the state diagram, you include in the bottom half of the state's rounded rectangle the phrase "do/" followed by the activity that is to be done while in that state. The do-activity may finish before any state transitions occur, after which the activity state behaves like a normal waiting state. If a transition out of the activity state occurs before the do-activity is finished, then the do-activity is interrupted.

Because a trigger event is optional when a transition occurs, it is possible that no trigger event may be listed as part of a transition's label. In such cases for normal waiting states, the object will immediately transition from that state to the new state. For activity states, such a transition is taken as soon as the do-activity finishes.

Figure A1.14 illustrates this situation using the states for a business telephone. When a caller is placed on hold, the call goes into the **On hold with music** state

FIGURE A1.14

A state diagram with an activity state and a triggerless transition



(soothing music is played for 10 seconds). After 10 seconds, the do-activity of the state is completed and the state behaves like a normal nonactivity state. If the caller pushes the # key when the call is in the **On hold with music** state, the call transitions to the **Canceled** state and then transitions immediately to the **dial tone** state. If the # key is pushed before the 10 seconds of soothing music has completed, the do-activity is interrupted and the music stops immediately.

Object Constraint Language—An Overview

The wide variety of diagrams available as part of UML provide you with a rich set of representational forms for the design model. However, graphical representations are often not enough. You may need a mechanism for explicitly and formally representing information that constrains some element of the design model. It is possible, of course, to describe constraints in a natural language such as English, but this approach invariably leads to inconsistency and ambiguity. For this reason, a more formal language—one that draws on set theory and formal specification languages (see Chapter 28 and Appendix 3) but has the somewhat less mathematical syntax of a programming language—seems appropriate.

The *Object Constraint Language* (OCL) complements UML by allowing you to use a formal grammar and syntax to construct unambiguous statements about various design model elements (e.g., classes and objects, events, messages, interfaces). The simplest OCL statements are constructed in four parts: (1) a *context* that defines the limited situation in which the statement is valid, (2) a *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute), (3) an *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and (4) keywords (e.g., **if, then, else, and, or, not, implies**) that are used to specify conditional expressions.

As a simple example of an OCL expression, consider the printing system discussed in Chapter 14. The guard condition placed on the **jobCostAccepted** event that causes a transition between the states *computingJobCost* and *formingJob* within the statechart diagram for the **PrintJob** object (Figure 14.9). In the diagram (Figure 14.9), the guard condition is expressed in natural language and implies that authorization can only occur if the customer is authorized to approve the cost of the job. In OCL, the expression may take the form:

```
customer
    self.authorizationAuthority = 'yes'
```

where a Boolean attribute, **authorizationAuthority,** of the class (actually a specific instance of the class) named **Customer** must be set to yes for the guard condition to be satisfied.

As the design model is created, there are often instances in which pre- or postconditions must be satisfied prior to completion of some action specified by the design. OCL provides a powerful tool for specifying pre- and postconditions in a formal manner. As an example, consider an extension to the print shop system (discussed as an example in Chapter 14) in which the customer provides an upper cost bound for the print job and a "drop-dead" delivery date at the same time as other print job characteristics are specified. If cost and delivery estimates exceed these bounds, the job is not submitted and the customer must be notified. In OCL, a set of pre- and postconditions may be specified in the following manner:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
    Integer)
 pre: upperCostBound > 0
    and custDeliveryReq > 07
    and self.jobAuthorization = 'no'
 post: if self.totalJobCost <= upperCostBound
       and self.deliveryDate <= custDeliveryReq
      then
       self.jobAuthorization = 'yes'
      endif
```

This OCL statement defines an invariant **(inv**)—conditions that must exist prior to (pre) and after (post) some behavior. Initially, a precondition establishes that bounding cost and delivery date must be specified by the customer, and authorization must be set to "no." After costs and delivery are determined, the postcondition specified is applied. It should also be noted that the expression:

```
self.jobAuthorization = 'yes'
```

is not assigning the value "yes" but is declaring that the **jobAuthorization** must have been set to "yes" by the time the operation finishes. A complete description of OCL is beyond the scope of this appendix. The complete OCL specification can be obtained at **www.omg.org/technology/documents/formal/ocl.htm.**[4]

Further Readings and Information Sources

Dozens of books discuss UML. Books by Hay (*UML and Data Modeling: A Reconciliation,* Technics Publications, 2011) and (*Enterprise Model Patterns: Describing the World,* Technics Publications, 2011), Gomaa (*Software Modeling and Design: UML, Use Cases, Patterns, and Software Architecture,* Cambridge University Press, 2011), (*Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, 2009), and Podesa (*UML for the IT Business Analyst*, Course Technology, 2009) discuss the use of UML in software development. Other books that provide useful information include: Miles and Hamilton (*Learning UML 2.0,* O'Reilly Media, 2006); Booch, Rumbaugh, and Jacobson (*Unified Modeling Language User Guide,* 2nd ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style,* Cambridge University Press, 2005), and Pilone and Pitman (*UML 2.0 in a Nutshell,* O'Reilly Media, 2005).

A wide variety of information sources on the use of UML in the software engineering modeling is available on the Internet. An up-to-date list of World Wide Web references can be found under "analysis" and "design" at the SEPA website: **www.mhhe.com/pressman.**

4 A description of the latest version of OCL can be found at **http://www.omg.org/spec/OCL/2.3.1/**

This page intentionally left blank

APPENDIX

Object-Oriented Concepts

2

Key Concepts

attributes 893
classes 892
boundary 894
characteristics of 897
controller 894
definition of . . . 892
entity 894
encapsulation 891
inheritance 894
messages 895
methods 893
operations 893
polymorphism 896
services 893
subclass 893
superclass. 893

What is an object-oriented (OO) viewpoint? Why is a method considered to be object oriented? What is an object? As OO concepts gained widespread adherents during the 1980s and 1990s, there were many different opinions about the correct answers to these questions, but today a coherent view of OO concepts has emerged. This appendix is designed to provide you with a brief overview of this important topic and to introduce basic concepts and terminology.

To understand the object-oriented point of view, consider an example of a real-world object—the thing you are sitting in right now—a chair. **Chair** is a subclass of a much larger class that we can call **PieceOfFurniture.** Individual chairs are members (usually called instances) of the class **Chair**. A set of generic attributes can be associated with every object in the class **PieceOfFurniture**. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible attributes. These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **Chair** is a member of **PieceOfFurniture**, **Chair** inherits all attributes defined for the class.

We have attempted an anecdotal definition of a class by describing its attributes, but something is missing. Every object in the class **PieceOfFurniture** can be manipulated in a variety of ways. It can be bought and sold, physically modified (e.g., you can saw off a leg or paint the object purple), or moved from one place to another. Each of these *operations* (other terms are *services* or *methods*) will modify one or more attributes of the object. For example, if the attribute **location** is a composite data item defined as

location = building + floor + room

then an operation named *move()* would modify one or more of the data items (**building, floor,** or **room**) that form the attribute **location**. To do this, *move* must have "knowledge" of these data items. The operation *move()* could be used for a chair or a table, as long as both are instances of the class **PieceOfFurniture**. Valid operations for the class **PieceOfFurniture**—*buy(), sell(), weigh()*—are specified as part of the class definition and are inherited by all instances of the class.

The class **Chair** (and all objects in general) encapsulates data (the attribute values that define the chair), operations (the actions that are applied to change the attributes of chair), other objects, constants (set values), and other related information. *Encapsulation* means that all of this information is packaged under one name and can be reused as one specification or program component.

Now that we have introduced a few basic concepts, a more formal definition of *object oriented* will prove more meaningful. Coad and Yourdon [Coa91] define the term this way:

Object oriented = objects + classification + inheritance + communication

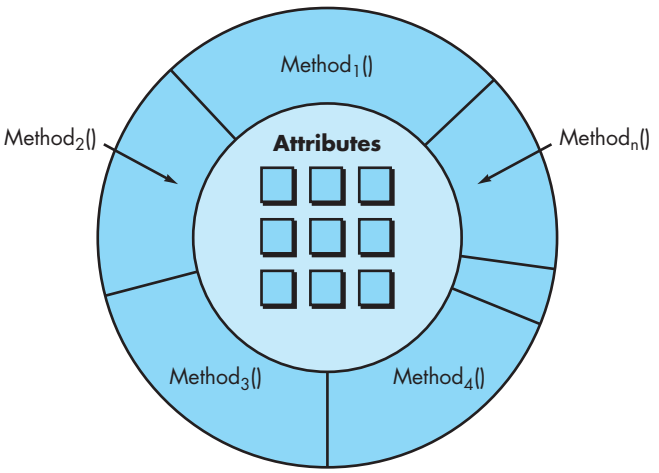
Three of these concepts have already been introduced. Communication is discussed later in this appendix.

Classes and Objects

A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity. Data abstractions that describe the class are enclosed by a "wall" of procedural abstractions [Tay90] (represented in Figure A2.1) that are capable of manipulating the data in some way. In a well-designed class, the only way to reach the attributes (and operate on them) is to go through one of the methods that form the "wall" illustrated in the figure. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding (Chapter 12) and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, their cohesion is improved, and because communication occurs only through the methods that make up the "wall," the class tends to be less strongly coupled from other elements of a system.[1]

Figure A2.1

A schematic representation of a class



1 It should be noted, however, that coupling can become a serious problem in OO systems. It arises when classes from various parts of the system are used as the data types of attributes, and arguments to methods. Even though access to the objects may only be through procedure calls, this does not mean that coupling is necessarily low, just lower than if direct access to the internals of objects were allowed.

Stated another way, a class is a generalized description (e.g., a template or blueprint) that describes a collection of similar objects. By definition, objects are instances of a specific class and inherit its attributes and the operations that are available to manipulate the attributes. A *superclass* (often called a *base class*) is a generalization of a set of classes that are related to it. A *subclass* is a specialization of the superclass. For example, the superclass **MotorVehicle** is a generalization of the classes **Truck, SUV, Automobile,** and **Van.** The subclass **Automobile** inherits all attributes of **MotorVehicle,** but in addition, incorporates other attributes that are specific only to automobiles.

These definitions imply the existence of a class hierarchy in which the attributes and operations of the superclass are inherited by subclasses that may each add additional "private" attributes and methods. For example, the operations *sitOn()* and *turn()* might be private to the **Chair** subclass.

Attributes

You have learned that attributes are attached to classes and that they describe the class in some way. An attribute can take on a value defined by an enumerated *domain*. In most cases, a domain is simply a set of specific values. For example, assume that a class **Automobile** has an attribute **color**. The domain of values for **color** is {**white, black, silver, gray, blue, red, yellow, green**}. In more complex situations, the domain can be a class. Continuing the example, the class **Automobile** also has an attribute **powerTrain** that is itself a class. The class **PowerTrain** would contain attributes that describe the specific engine and transmission for the car.

The *features* (values of the domain) can be augmented by assigning a default value (feature) to an attribute. For example, the **color** attribute defaults to **white**. It may also be useful to associate a probability with a particular feature by assigning {value, probability} pairs. Consider the **color** attribute for automobile. In some applications (e.g., manufacturing planning) it might be necessary to assign a probability to each of the colors (e.g., white and black are highly probable as automobile colors).

Operations, Methods, and Services

An object encapsulates data (represented as a collection of attributes) and the algorithms that process the data. These algorithms are called *operations, methods,* or *services*[2] and can be viewed as processing components.

Each of the operations that is encapsulated by an object provides a representation of one of the behaviors of the object. For example, the operation *GetColor()* for the object **Automobile** will extract the color stored in the **color** attribute. The implication of the existence of this operation is that the class **Automobile** has

2 In the context of this discussion, the term *operations* is used, but the terms *methods* and *services* are equally popular.

been designed to receive a stimulus (we call the stimulus a *message*) that requests the color of the particular instance of a class. Whenever an object receives a stimulus, it initiates some behavior. This can be as simple as retrieving the color of automobile or as complex as the initiation of a chain of stimuli that are passed among a variety of different objects. In the latter case, consider an example in which the initial stimulus received by **Object 1** results in the generation of two other stimuli that are sent to **Object 2** and **Object 3**. Operations encapsulated by the second and third objects act on the stimuli, returning necessary information to the first object. **Object 1** then uses the returned information to satisfy the behavior demanded by the initial stimulus.

Object-Oriented Analysis and Design Concepts

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These *entity classes* typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design. *Boundary classes* create the interface (e.g., interactive screen and printed reports) that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Controller classes are designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, and (4) validation of data communicated between objects or between the user and the application.

The concepts discussed in the paragraphs that follow can be useful in analysis and design work.

Inheritance. Inheritance is one of the key differentiators between conventional and object-oriented systems. A subclass **Y** inherits all of the attributes and operations associated with its superclass **X**. This means that all data structures and algorithms originally designed and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly.

Any change to the attributes or operations contained within a superclass is immediately inherited by all subclasses. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system.

It is important to note that at each level of the class hierarchy new attributes and operations may be added to those that have been inherited from higher

levels in the hierarchy. In fact, whenever a new class is to be created, you have a number of options:

- The class can be designed and built from scratch. That is, inheritance is not used.
- The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.
- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
- Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class.

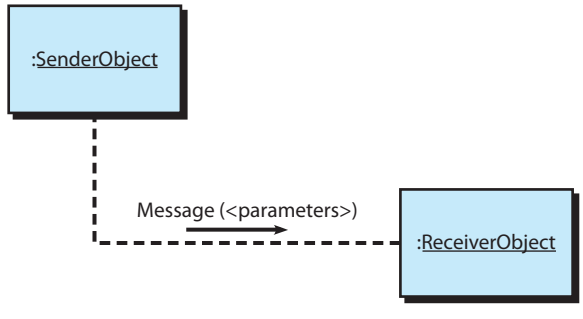
Like all fundamental design concepts, inheritance can provide significant benefit for the design, but if it is used inappropriately,[3] it can complicate a design unnecessarily and lead to error-prone software that is difficult to maintain.

Messages. Classes must interact with one another to achieve design goals. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

The interaction between objects is illustrated schematically in Figure A2.2. An operation within **SenderObject** generates a message of the form *message* (<parameters>) where the parameters identify **ReceiverObject** as the object to be stimulated by the message, the operation within **ReceiverObject** that is to receive the message, and the data items that provide information that is required for the operation to be successful. The collaboration defined between classes as part of the analysis model provides useful guidance in the design of messages.

FIGURE A2.2

Message passing between objects



3 For example, designing a subclass that inherits attributes and operations from more than one superclass (sometimes called "multiple inheritance") is frowned upon by most designers.

Cox [Cox86] describes the interchange between classes in the following manner:

> An object [class] is requested to perform one of its operations by sending it a message telling the object what to do. The receiver [object] responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller. Messaging ties an object-oriented system together. Messages provide insight into the behavior of individual objects and the OO system as a whole.

Polymorphism. *Polymorphism* is a characteristic that greatly reduces the effort required to extend the design of an existing object-oriented system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself. To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph. Then, within the design, control logic similar to the following would have to be embedded:

```
case of graphtype:
    if graphtype = linegraph then DrawLineGraph (data);
    if graphtype = piechart then DrawPieChart (data);
    if graphtype = histogram then DrawHisto (data);
    if graphtype = kiviat then DrawKiviat (data);
end case;
```

Although this design is reasonably straightforward, adding new graph types could be tricky. A new drawing module would have to be created for each graph type and then the control logic would have to be updated to reflect the new graph type.

To solve this problem, all of the graphs become subclasses of a general class called **Graph**. Using a concept called *overloading* [Tay90], each subclass defines an operation called *draw*. An object can send a *draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph. Therefore, the design is reduced to

```
draw <graphtype>
```

When a new graph type is to be added to the system, a subclass is created with its own *draw* operation. But no changes are required within any object that wants a graph drawn because the message **draw <graphtype>** remains unchanged. To summarize, polymorphism enables a number of different operations to have the same name. This in turn decouples objects from one another, making each more independent.

Design classes. The requirements model defines a complete set of analysis classes. Each describes some element of the problem domain, focusing on

aspects of the problem that are user or customer visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, the software team must define a set of *design classes* that (1) refine the analysis classes by providing design detail that will enable the classes to be implemented and (2) create a new set of design classes that implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture are suggested [Amb01]:

- *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI).
- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architectural design evolves, the software team should develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent objects (and associated methods that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness. Methods associated with a design class should be focused on accomplishing one specific function for the class. Once the function has

been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip** of the video editing software might have attributes **start-point** and **end-point** to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, *setStartPoint()* and *setEndPoint()* provide the only means for establishing start and end points for the clip.

High cohesion. A cohesive design class is single minded. That is, it has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** of the video-editing software might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, test, and maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.[4]

Further Readings and Information Sources

Over the past three decades hundreds of books have been written on object-oriented programming, analysis, and design. Weisfeld (*The Object-Oriented Thought Process,* 4th ed., Addison-Wesley, 2013) presents a worthwhile treatment of general OO concepts and principles. Wong and Nguyen *(Principles of Object-Oriented Programming,* Amazon Digital Services, 2011) provide a practical tutorial on many important OO concepts. McLaughlin and his colleagues (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D,* O'Reilly Media, 2006) have written an accessible and enjoyable treatment of OO analysis and design approaches. Keogh and Gianni (*OOP Demystified*, McGraw-Hill, 2004) offer a useful guidebook on the subject.

A more in-depth treatment of OO analysis and design is presented by Booch and his colleagues (*Object-Oriented Analysis and Design with Applications,* 3rd ed., Addison-Wesley, 2007). Clark (*Beginning C# Object-Oriented Programming,* Apress, 2013), Kochen (*Programming in Objective-C,* 5th ed., Developer's Library, 2012), Phillips (*Python 3 Object Oriented Programming,* Packt Publishing, 2011), Khurana (*Object-Oriented Programming with C++,* Vikas Publishing, 2010), and Wu (*An Introduction to Object-Oriented Programming with Java,* 2d ed., McGraw-Hill, 2009) are representative of dozens of OO titles written for many different programming languages.

A wide variety of information sources on object-oriented technologies is available on the Internet. An up-to-date list of World Wide Web references can be found under "analysis" and "design" at the SEPA website: **www.mhhe.com/pressman.**

4 A less formal way of stating the law of Demeter is, "Each unit should only talk to its friends; don't talk to strangers."

APPENDIX

FORMAL METHODS

3

KEY CONCEPTS

formal specification language 900
Object Constraint Language (OLC). . . 901
notation. 902
example of . . . 903
Z-specification language 904
notation. 905
example of. . . . 906

APPLYING MATHEMATICAL NOTATION[1] FOR FORMAL SPECIFICATION

To illustrate the use of mathematical notation in the formal specification of a software component, we revisit the block handler example presented in Chapter 28. The system for managing blocks is depicted schematically in Figure 28.8 and should be reviewed before continuing here.

A set named *BLOCKS* will consist of every block number. *AllBlocks* is a set of blocks that lie between 1 and *MaxBlocks*. The state will be modeled by two sets and a sequence. The two sets are *used* and *free*. Both contain blocks—the *used* set contains the blocks that are currently used in files, and the *free* set contains blocks that are available for new files. The sequence will contain sets of blocks that are ready to be released from files that have been deleted. The state can be described as

$$used, free : \mathbb{P}\ BLOCKS$$
$$BlockQueue : seq\ \mathbb{P}\ BLOCKS$$

This is very much like the declaration of program variables. It states that *used* and *free* will be sets of blocks and that *BlockQueue* will be a sequence, each element of which will be a set of blocks. The data invariant can be written as

$$used \cap free = \varnothing \wedge$$
$$used \cup free = AllBlocks \wedge$$
$$\forall\ i : \mathrm{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq used \wedge$$
$$\forall\ i, j : \mathrm{dom}\ BlockQueue \bullet i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

The first line of the data invariant states that there will be no common blocks in the used collection and free collections of blocks. The second line states that the collection of used blocks and free blocks will always be equal to the whole collection of blocks in the system. The third line indicates the *i*th element in the block queue will always be a subset of the used blocks. The final line states that, for any two elements of the block queue that are not the same, there will be no common blocks in these two elements.

1 We have written this section of Appendix 3 making the assumption that you are familiar with the mathematical notation associated with sets and sequences and the logical notation used in predicate calculus. If you need a review, a brief overview is presented as a supplementary resource at the 8th edition website. For more detailed information, see [Jec06] or [Pot04].

The first operation to be defined is one that removes an element from the head of the block queue. The precondition is that there must be at least one item in the queue:

$$\#\mathrm{BlockQueue} > 0,$$

The postcondition is that the head of the queue must be removed and placed in the collection of free blocks and the queue adjusted to show the removal:

$$\begin{aligned}
&used' = used \setminus head\ BlockQueue \wedge \\
&free' = free \cup head\ BlockQueue \wedge \\
&BlockQueue' = tail\ BlockQueue
\end{aligned}$$

A convention used in many formal methods is that the value of a variable after an operation is primed. Hence, the first component of the preceding expression states that the new used blocks (*used'*) will be equal to the old used blocks minus the blocks that have been removed. The second component states that the new free blocks (*free'*) will be the old free blocks with the head of the block queue added to it. The third component states that the new block queue will be equal to the tail of the old value of the block queue, that is, all elements in the queue apart from the first one. A second operation adds a collection of blocks, *Ablocks,* to the block queue. The precondition is that *Ablocks* is currently a set of used blocks:

? How do I represent pre- and postconditions?

$$Ablocks \subseteq used$$

The postcondition is that the set of blocks is added to the end of the block queue and the set of used and free blocks remains unchanged:

$$\begin{aligned}
&BlockQueue' = BlockQueue \frown \langle \mathrm{Ablocks} \rangle \wedge \\
&used' = used \wedge \\
&free' = free
\end{aligned}$$

There is no question that the mathematical specification of the block queue is considerably more rigorous than a natural language narrative or a graphical model. The additional rigor requires effort, but the benefits gained from improved consistency and completeness can be justified for some application domains.

Formal Specification Languages

A formal specification language is usually composed of three primary components: (1) a syntax that defines the specific notation with which the specification is represented, (2) semantic domain to help define a "universe of objects" [Win90] that will be used to describe the system, and (3) a set of relations that define the semantic rules that indicate how objects may be manipulated properly and satisfy the specification.

The syntactic domain of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus. The *semantic domain* of a specification language indicates how the language represents system requirements.

A variety of formal specification languages are in use today. OCL [OMG03b], Z [ISO02], LARCH [Gut93], and VDM [Jon91] are representative formal specification languages that exhibit the characteristics noted previously. In this appendix, we present a brief discussion of OCL and Z.

Object Constraint Language (OCL)[2]

Object Constraint Language (OCL) is a formal notation developed so that users of UML can add more precision to their specifications. All of the power of logic and discrete mathematics is available in the language. However, the designers of OCL decided that only ASCII characters (rather than conventional mathematical notation) should be used in OCL statements.

To use OCL, you start with one or more UML diagrams—most commonly class, state, or activity diagrams (Appendix 1). OCL expressions are added and state facts about elements of the diagrams. These expressions are called *constraints*; any implementation derived from the model must ensure each of the constraints always remains true.

Like an object-oriented programming language, an OCL expression involves operators operating on objects. However, the result of a complete expression must always be a Boolean, that is, true or false. The objects can be instances of the OCL **Collection** class, of which **Set** and **Sequence** are two subclasses.

The object **self** is the element of the UML diagram in the context of which the OCL expression is being evaluated. Other objects can be obtained by *navigating* using the . (dot) symbol from the **self** object. For example:

- If **self** is class **C**, with attribute **a**, then **self.a** evaluates to the object stored in **a**.
- If **C** has a one-to-many association called *assoc* to another class **D**, then **self.assoc** evaluates to a **Set** whose elements are of type **D**.
- Finally (and a little more subtly), if **D** has attribute **b**, then the expression **self.assoc.b** evaluates to the set of all the **b**s belonging to all **D**s.

OCL provides built-in operations implementing set and logic operators, constructive specification, and related mathematics. A small sample of these is presented in Table A3.1.

2 This section has been contributed by Professor Timothy Lethbridge of the University of Ottawa and is presented here with permission.

TABLE A3.1 SUMMARY OF KEY OCL NOTATION

x.y	Obtain the property y of object x. A property can be an attribute, the set of objects at the end of an association, the result of evaluating an operation, or other things depending on the type of UML diagram. If x is a Set, then y is applied to every element of x; the results are collected into a new Set.
c–>f()	Apply the built-in OCL operation f to Collection c itself (as opposed to each of the objects in c). Examples of built-in operations are listed below.
and, or, =, <>	Logical and, logical or, equals, not-equals.
p implies q	True if either q is true or p is false.
Sample of Operations on Collections (including Sets and Sequences)	
C–>size()	The number of elements in Collection c.
C–>isEmpty()	True if c has no elements, false otherwise.
c1–>includesAll(c2)	True if every element of c2 is found in c1.
c1–>excludesAll(c2)	True if no element of c2 is found in c1.
C–>forAll(elem | boolexpr)	True if boolexpr is true when applied to every element of c. As an element is being evaluated, it is bound to the variable elem, which can be used in boolexpr. This implements universal quantification, discussed earlier.
C–>forAll(elem1, elem2 | boolexpr)	Same as above, except that boolexpr is evaluated for every possible *pair* of elements taken from c, including cases where the pair consists of the same element.
C–>isUnique(elem |expr)	True if expr evaluates to a different value when applied to every element of c.
Sample of Operations Specific to Sets	
s1–>intersection(s2)	The set of those elements found s1 and also in s2.
s1–>union(s2)	The set of those elements found in either s1 or s2.
s1–>excluding(x)	The set s1 with object x omitted.
Sample Operation Specific to Sequences	
Seq–>first()	The object that is the first element in the sequence seq.

To illustrate the use of OCL in specification, we reexamine the block handler example, introduced in Chapter 28. The first step is to develop a UML model (Figure A3.1). This class diagram specifies many relationships among the objects involved. However, OCL expressions are added to allow implementers of the system to know more precisely what must remain true as the system runs.

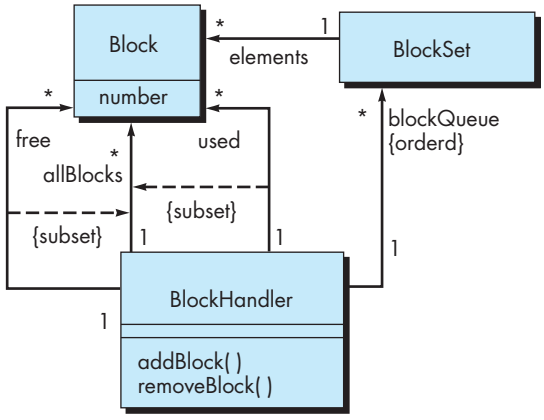
The OCL expressions that supplement the class diagram correspond to the six parts of the invariant discussed in Section 28.6. In the example that follows, the invariant is repeated in English and then the corresponding OCL expression is written. It is considered good practice to provide natural language text along with the formal logic; doing so helps you to understand the logic, and also helps reviewers to uncover mistakes, for example, situations where English and the logic do not correspond.

1. *No block will be marked as both unused and used.*

```
context BlockHandler inv:
    (self.used–>intersection(self.free)) –>isEmpty0
```

FIGURE A3.1

Class diagram for a block handler



Note that each expression starts with the keyword ***context***. This indicates the element of the UML diagram that the expression constrains. The keyword **self** here refers to the instance of **BlockHandler**; in the following, as is permissible in OCL, we will omit the **self**.

2. *All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.*

```
context BlockHandler inv:
    blockQueue–>forAll(aBlockSet | used–>includesAll(aBlockSet ))
```

3. *No elements of the queue will contain the same block numbers.*

```
context BlockHandler inv:
    blockQueue–>forAll(blockSet1, blockSet2 |
        blockSet1 <> blockSet2 implies
        blockSet1.elements.number–>excludesAll(blockSet2.elements.number))
```

The expression before **implies** is needed to ensure we ignore pairs where both elements are the same block.

4. *The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.*

```
context BlockHandler inv:
    allBlocks = used–>union(free)
```

5. *The collection of unused blocks will have no duplicate block numbers.*

```
context BlockHandler inv:
    free–>isUnique(aBlock | aBlock.number)
```

6. *The collection of used blocks will have no duplicate block numbers.*

```
context BlockHandler inv:
    used–>isUnique(aBlock | aBlock.number)
```

OCL can also be used to specify preconditions and postconditions of operations. For example, the following describes operations that remove and add sets of blocks to the queue. Note that the notation **x@pre** indicates the object **x** as it existed *prior* to the operation; this is opposite to mathematical notation discussed earlier, where it is the **x** *after* the operation that is specially designated (as **x**′).

```
context BlockHandler::removeBlocks( )
    pre: blockQueue–>size( ) >0
    post: used = used@pre-blockQueue@pre–>first( ) and
        free = free@pre–>union(blockQueue@pre–>first( )) and
        blockQueue = blockQueue@pre–>excluding(blockQueue@pre–>first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
    pre: used–>includesAll(aBlockSet.elements)
    post: (blockQueue.elements = blockQueue.elements@pre
        –> append (aBlockSet.elements) and
        used = used@pre and
        free = free@pre
```

OCL is a modeling language, but it has all of the attributes of a formal language. OCL allows the expression of various constraints, pre- and postconditions, guards, and other characteristics that relate to the objects represented in various UML models.

The Z Specification Language

Z (properly pronounced as "zed") is a specification language that is widely used within the formal methods community. The Z language applies typed sets, relations, and functions within the context of first-order predicate logic to build *schemas*—a means for structuring the formal specification.

Z specifications are organized as a set of schemas. Schemas are used to structure a formal specification in the same way that components are used to structure a system.

A schema describes the stored data that a system accesses and alters. In the context of Z, this is called the "state." This usage of the term *state* in Z is slightly different from the use of the word in the rest of this book.[3] The generic structure of a schema takes the form:

```
———schemaName————————————————————————
    declarations
————————————————————————————————————
    invariant
————————————————————————————————————
```

3 Recall that in other chapters *state* has been used to identify an externally observable mode of behavior for a system.

where declarations identify the variables that comprise the system state and the invariant imposes constraints on the manner in which the state can evolve. A summary of Z language notation is presented in Table A3.2.

TABLE A3.2 SUMMARY OF Z NOTATION

Z notation is based on typed set theory and first-order logic. Z provides a construct, called a schema, to describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the possible value of a variable. In Z, the schema X is defined by the form

```
———X———————————————————————
    declarations
———————————————————————————
    predicates
———————————————————————————
```

Global functions and constants are defined by the form

```
    declarations
———————————————————————————
    predicates
```

The declaration gives the type of the function or constant, while the predicate gives it value. Only an abbreviated set of Z symbols is presented in this table.

Sets:

$S : \mathbb{P}X$	S is declared as a set of Xs.
$X \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ….
$S : \mathbb{F}\, X$	S is declared as a finite set of Xs.
max (S)	The maximum of the nonempty set of numbers S.

Functions:

$f{:}X \rightarrowtail Y$	f is declared as a partial injection from X to Y.
dom f	The domain of f: the set of values x for which $f(x)$ is defined.
ran f	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \trianglelefteq f$	A function like f, except that x is removed from its domain.

Logic:

$P \wedge Q$	P and Q: It is true if both P and Q are true.
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false.
$\theta S' = \theta S$	No components of schema S change in an operation.

The following example of a schema describes the state of the block handler and the data invariant:

———**BlockHandler**———

$used, free : \mathbb{P}\ BLOCKS$
$BlockQueue : seq\ \mathbb{P}\ BLOCKS$
$used \cap free = \varnothing \wedge$
$used \cup free = AllBlocks \wedge$
$\forall i: \mathbf{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq used \wedge$
$\forall i, j: \mathrm{dom}\ BlockQueue \bullet i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \varnothing$

As we have noted, the schema consists of two parts. The part above the central line represents the variables of the state, while the part below the central line describes the data invariant. Whenever the schema specifies operations that change the state, it is preceded by the Δ symbol. The following example of a schema describes the operation that removes an element from the block queue:

———RemoveBlocks———

$\Delta\ BlockHandler$

$\#\mathrm{BlockQueue} > 0,$
$used' = used \setminus head\ BlockQueue \wedge$
$\mathrm{free}' = free \cup head\ BlockQueue \wedge$
$BlockQueue' = tail\ BlockQueue$

The inclusion of Δ *BlockHandler* results in all variables that make up the state being available for the *RemoveBlocks* schema and ensures that the data invariant will hold before and after the operation has been executed.

The second operation, which adds a collection of blocks to the end of the queue, is represented as

———AddBlocks———

$\Delta\ \mathrm{BlockHandler}$
$\mathrm{Ablocks?} : \mathrm{BLOCKS}$

$Ablocks? \subseteq used$
$BlockQueue' = BlockQueue \frown \langle \mathrm{Ablocks?} \rangle \wedge$
$used' = used \wedge$
$free' = free$

By convention in Z, an input variable that is read, but does not form part of the state, is terminated by a question mark. Thus, Ablocks?, which acts as an input parameter, is terminated by a question mark.

Further Readings and Information Sources

In the formal methods domain, books by Gabbar (*Modern Formal Methods and Applications*, Springer, 2010), Casey (*A Programming Approach to Formal Methods,* McGraw-Hill, 2000), Hinchey and Bowan (*Industrial Strength Formal Methods,* Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997), and Sheppard (*An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995) provide useful guidance. In addition, language-specific books such as Liu (*Formal Engineering for Industrial Software Development: Using SOFL Method, 2010*), Warmer and Kleppe (*The Object Constraint Language: Precise Modeling with UML,* Addison-Wesley, 2005) and (*Object Constraint Language,* Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods,* Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z,* Wiley, 1997), and Cooper and Barden (*Z in Practice,* Prentice Hall, 1995) provide useful introductions to formal methods as well as a variety of modeling languages.

A wide variety of information sources on formal methods is available on the Internet. An up-to-date list of World Wide Web references can be found under "software engineering resources" at the SEPA website: **www.mhhe.com/pressman.**

This page intentionally left blank

References

[Abb83] Abbott, R., "Program Design by Informal English Descriptions, "*CACM*, vol. 26, no. 11, November 1983, pp. 892–894.

[Abr09] Abrial, J., "Faultless Systems: Yes We Can!" *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 30–36.

[ACM12] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 2012, available at www.acm.org/serving/se/code.htm.

[Ada93] Adams, D., *Mostly Harmless*, Macmillan, 1993.

[AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-45, U.S. Air Force, September 30, 1988.

[Agi03] The Agile Alliance Home Page, available at www.agilealliance.org/home.

[Air99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.

[Aka04] Akao, Y., *Quality Function Deployment*, Productivity Press, 2004.

[Ale11] Alexander, I. "Gore, Sore, or What?" *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 8–10.

[Ale77] Alexander, C., *A Pattern Language*, Oxford University Press, 1977.

[Ale79] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.

[All08] Allen, J. H., et al., *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley, 2008.

[Amb01] Ambler, S., *The Object Primer*, 2nd ed., Cambridge University Press, 2001.

[Amb02a] Ambler, S., "What Is Agile Modeling (AM)?" 2002, available at www.agilemodeling.com/index.htm.

[Amb02b] Ambler, S., and R. Jeffries, *Agile Modeling*, Wiley, 2002.

[Amb02c] Ambler, S., "UML Component Diagramming Guidelines," 2002, available at www.modelingstyle.info/.

[Amb04] Ambler, S., "*Examining the Cost of Change Curve*," in *The Object Primer*, 3rd ed., Cambridge University Press, 2004.

[Amb06] Ambler, S., "The Agile Unified Process (AUP), 2006, available at www.ambysoft.com/unifiedprocess/agileUP.html.

[Amb95] Ambler, S., "Using Use-Cases," *Software Development*, July 1995, pp. 53–61.

[Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.

[And05] Andreou, A., et al., "Key Issues for the Design and Development of Mobile Commerce Services and Applications," *International Journal of Mobile Communications*, vol. 3, no. 3, March 2005, pp. 303–323.

And06] Andrews, M., and J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley, 2006.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.

[Ant06] Anton, D., and C. Anton, *ISO 9001 Survival Guide*, 3rd ed., AEM Consulting Group, 2006.

[AOS07] AOSD.net (Aspect-Oriented Software Development), glossary, available at http://aosd.net/wiki/index.php?title=Glossary.

[App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," February 2000, available at www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[App13] Apple Computer, *Accessibility*, 2013, available at www.apple.com/accessibility/.

[Arl02] Arlow, J., and I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.

[Arn89] Arnold, R. S., "Software Restructuring," *Proceedings of the IEEE*, vol. 77, no. 4, April 1989, pp. 607–617.

[Art97] Arthur, L. J., "Quantum Improvements in Software System Quality," *CACM*, vol. 40, no. 6, June 1997, pp. 47–52.

[Ast04] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2004.

[ATK12] ATKearney, "A.T. Kearney Study of Global Wealth and Spending," 2012, available at http://www.atkearney.com/news-media/news-releases/news-release/-/asset_publisher/00OIL7Jc67KL/content/id/387464.

[Baa07] de Baar, B., "Project Risk Checklist," 2007, available at www.softwareprojects.org/project_riskmanagement_starting62.htm.

[Baa10] Baaz, A., et al., "Appreciating Lessons Learned," *IEEE Software*, vol. 27, no. 4, July–August, 2010, pp. 72–79.

[Bab09] Babar, M., and I. Groton, "Software Architecture Review: The State of Practice," *IEEE Computer*, vol. 42, no. 6, June 2009, pp. 1–8.

[Bab86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.

[Bac97] Bach, J., "'Good Enough Quality: Beyond the Buzzword," *IEEE Computer*, vol. 30, no. 8, August 1997, pp. 96–98.

[Bac98] Bach, J., "The Highs and Lows of Change Control," *Computer*, vol. 31, no. 8, August 1998, pp. 113–115.

[Bae98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.

[Bak72] Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal.*, vol. 11, no. 1, 1972, pp. 56–73.

[Ban06a] Baniassad, E., et al., "Discovering Early Aspects," *IEEE Software*, vol. 23, no. 1, January–February, 2006, pp. 61–69.

[Bar06b] Baresi, L., E. DiNitto, and C. Ghezzi, "Toward Open-World Software: Issues and Challenges," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 36–43.

[Bas03] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003

[Bas84] Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728–738.

[Bea11] Beaird, J., The Principles of Beautiful Web Design, 2nd ed., Sitepoint, 2011.

[Bec00] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Bec01] Beck, K., et al., "Manifesto for Agile Software Development," www.agilemanifesto.org/.

[Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2004.

[Bec04b] Beck, K., *Test-Driven Development: By Example*, 2nd ed., Addison-Wesley, 2004.

[Bee99] Beedle, M., et al., "SCRUM: An Extension Pattern Language for Hyperproductive Software Development," included in: *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading MA, 1999, downloadable from http://jeffsutherland.com/scrum/scrum_plop.pdf.

[Beg10] Begel, A., R. DeLine, and T. Zimmermann, "Social Media for Software Engineering," *Proc. FoSER 2010*, ACM, November, 2010.

[Bei84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.

[Bei90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.

[Bei95] Beizer, B., *Black-Box Testing*, Wiley, 1995.

[Bel81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L. J. Peters, author), Yourdon Press, 1981.

[Bel95] Bellinzona R., M. G. Gugini, and B. Pernici, "Reusing Specifications in OO Applications," *IEEE Software*, March 1995, pp. 65–75.

[Ben00] Bennatan, E., *Software Project Management: A Practitioner's Approach*, 3rd ed., McGraw-Hill, 2000.

[Ben99] Bentley, J., *Programming Pearls*, 2nd ed., Addison-Wesley, 1999.

[Ben02] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Analysis and Design*, 2nd ed., McGraw-Hill, 2002.

[Ben10] Benaroch, M., and A. Appari, "Financial Pricing of Software Development Risk Facotrs," *IEEE Software*, vol. 27, no. 3, September–October, 2010, pp. 65–73.

[Ber80] Bersoff, E., V. Henderson, and S. Siegel, *Software Configuration Management*, Prentice Hall, 1980.

[Ber93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley, 1993.

[Bes04] Bessin, J., "The Business Value of Quality," IBM developerWorks, June 15, 2004, downloadable from http://cm.techwell.com/articles/original/business-value-quality-and-testing.

[Bie94] Bieman, J. M., and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Engineering*, vol. SE-20, no. 8, August 1994, pp. 308–320.

[Bin93] Binder, R., "Design for Reuse Is for Real," *American Programmer*, vol. 6, no. 8, August 1993, pp. 30–37.

[Bin94a] Binder, R., "Testing Object-Oriented Systems: A Status Report," *American Programmer*, vol. 7, no. 4, April 1994, pp. 23–28.

[Bin94b] Binder, R. V., "Object-Oriented Software Testing," *Communications of the ACM*, vol. 37, no. 9, September 1994, p. 29.

[Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[Bir98] Biró, M., and T. Remzsö, "Business Motivations for Software Process Improvement," ERCIM News No. 32, January 1998, available at www.ercim.org/publication/Ercim_News/enw32/biro.html.

[Bla09] Black, S., et al. "Formal Versus Agile: Survival of the Fittest?" *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 37–45.

[Bla10] Blair, S., et al., "Responsibility-Driven Architecture," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 26–32.

[Bod09] Bode, S., et al., "Software Architectural Design Meets Security Engineering," *Proceedings of 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2009.

[Boe00] Boehm, B., et al., *Software Cost Estimation in COCOMO II*, Prentice Hall, 2000.

[Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition," *CrossTalk*, May 2001, available at www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.

[Boe01b] Boehm, B., and V. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer*, vol. 34, no. 1, January 2001, pp. 135–137.

[Boe08] Boehm, B., "Making a Difference in the Software Century," *IEEE Computer*, vol. 41, no. 3, March 2008, pp. 32–38.

[Boe81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer*, vol. 21, no. 5, May 1988, pp. 61–72.

[Boe89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.

[Boe96] Boehm, B., "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 73–82.

[Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer*, vol. 31, no. 7, July 1998, pp. 33–44.

[Boh00] Bohl, M., and M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic*, 5th ed., Prentice Hall, 2000.

[Boh66] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *CACM*, vol. 9, no. 5, May 1966, pp. 366–371.

[Boi04] Boiko, B., *Content Management Bible*, 2nd ed., Wiley, 2004.

[Boo02] Booch, G., and A. Brown, "Collaborative Development Environments," Rational Software Corp., October 28, 2002.

[Boo05] Booch, G., J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*. 2nd ed., Addison-Wesley, 2005.

[Boo06] Bootstrap-institute.com, 2006, www.cse.dcu.ie/espinode/directory/directory.html.

[Boo08] Booch, G., *Handbook of Software Architecture*, 2008, available at www.booch.com/architecture/systems.jsp.

[Boo11a] Booch, G., "Dominant Design," *IEEE Software*, vol. 28, no. 2, January–February 2011, pp. 8–9.

[Boo11b] Booch, G., "Draw Me a Picture," *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 6–7.

[Boo94] Booch, G., *Object-Oriented Analysis and Design*, 2nd ed., Benjamin Cummings, 1994.

[Bor01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.

[Bos00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[Bos11] Bose, B., et al., "Morphing Smartphones into Automotive Application Platforms," *IEEE Computer*, vol. 44, no. 5, May 2011, pp. 28–29.

[Bra94] Bradac, M., D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Trans. Software Engineering*, vol. 20, no. 10, October 1994, pp. 774–784.

[Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software," informit.com, February 1, 2002, available at www.informit.com/articles/article.asp?p=25141&rl=1.

[Bre03] Breu, R., et al., *Key Issues of a Formally Based Process Model for Security Engineering*, Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications, *2003*.

[Bre10] Breu, R., *Ten Principles for Living Models—A Manifesto of Change-Driven Software Engineering*, International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), Krakow, Poland, February 2010, pp. 1–8.

[Bro01] Brown, B., *Oracle9i Web Development*, 2nd ed., McGraw-Hill, 2001.

[Bro03] Brooks, F., "Three Great Challenges for Half-Century-Old Computer Science," *JACM*, vol. 50, no. 1, January 2003, pp. 25–26.

[Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 72–80.

[Bro10a] Brown, N., R. Nord, and I. Ozkaya, "Enabling Agility through Architecture," *Crosstalk*, November–December 2010, available at www.crosstalkonline.org/storage/issue-archives/…/201011-Brown.pdf.

[Bro10b] Broy, M., and R. Reussner, "Software Architecture Review: The State of Practice," *IEEE Computer*, vol. 43, no. 10, June 2010, pp. 88–91.

[Bro11] Brown, M., "The Best Tools for Mobile App Testing," available at http://www.mobileapptesting.com/the-best-tools-for-mobile-app-testing/2011/08/.

[Bro12] Brown, A., *The Architecture of Open Source Applications*, lulu.com, 2012.

[Bro95] Brooks, F., *The Mythical Man-Month*, Silver Anniversary edition, Addison-Wesley, 1995.

[Bro96] Brown, A., and K. Wallnau, "Engineering of Component Based Systems," *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7–15.

[Buc99] Bucanac, C., "The V-Model," University of Karlskrona/Ronneby, January 1999, downloadable from www.bucanac.com/documents/The_V-Model.pdf.

[Bud96] Budd, T., *An Introduction to Object-Oriented Programming*, 2nd ed., Addison-Wesley, 1996.

[Bus07] Buschmann, F., et al., *Pattern-Oriented Software Architecture, A System of Pattern*, Wiley, 2007.

[Bus10] Buschmann, F., "Learning from Failure, Part 2: Featuritis, Performitis, and Other Diseases," *IEEE Software*, vol. 27, no. 1, January–February 2010, pp. 10–11.

[Bus10a] Buschmann, F., "On Architecture Styles and Paradigms," *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 92–94.

[Bus10b] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 1," *IEEE Software*, vol. 27, no. 3, May–June 2010, pp. 63–65.

[Bus10c] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 2," *IEEE Software*, vol. 27, no. 4, July–August 2010, pp. 12–14.

[Bus96] Buschmann, F., et al., *Pattern-Oriented Software Architecture*, Wiley, 1996.

[Cac02] Cachero, C., et al., "Conceptual Navigation Analysis: a Device and Platform Independent Navigation Specification," *Proceedings of the Second International Workshop on Web-Oriented Technology*, June 2002, available at www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.

[Car08] Carrasco, M., "7 Key Attributes of High Performance Software Development Teams," June 30, 2008, available at http://www.realsoftwaredevelopment.com/7-key-attributes-of-high-performance-software-development-teams/.

[Car90] Card, D., and R. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.

[Cas06] Casey, V., and I. Richardson, "Uncovering the Reality within Virtual Software Teams," *Proc. GSD'06*, ACM, May 23, 2006.

[Cas89] Cashman, M., "Object Oriented Domain Analysis," *ACM Software Engineering Notes*, vol. 14, no. 6, October 1989, p. 67.

[Cav78] Cavano, J., and J. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop*, November 1978, pp. 133–139.

[CCS02] CS3 Consulting Services, 2002, available at www.cs3inc.com/DSDM.htm.

[Cec06] Cechich, A., et al., "Trends on COTS Component Identification," *Proc. Fifth Intl. Conf. on COTS-Based Software Systems*, IEEE, 2006.

[Cha89] Charette, R., *Engineering Risk Analysis and Management*, McGraw-Hill/Intertext, 1989.

[Cha92] Charette, R., "Building Bridges over Intelligent Rivers," *American Programmer*, vol. 5, no. 7, September 1992, pp. 2–9.

[Cha93] de Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[Chi94] Chidamber, S., and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 476–493.

[Cho89] Choi, S., and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proceedings of the Second International Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 66–75.

[Chu09] Chung, L., and J. Leite, "On Non-Functional Requirements in Software Engineering," published in *Conceptual Modeling: Foundations and Applications*, Springer-Verlag, 2009.

[Chu95] Churcher, N., and M. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 69–76.

[Cia09] Ciafrani, C., et al., *ISO 9000:2008 Explained*, 3rd ed., ASQ Quality Press, 2009.

[Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings," 2007, available at www.cigital.com/solutions/roi-cs2.php.

[Cla05] Clark, S., and E. Baniasaad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.

[Cle03] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2003.

[Cle06] Clemmons, R., "Project Estimation with Use Case Points," *CrossTalk*, February 2006, pp. 18–222, available at www.stsc.hill.af.mil/crosstalk/2006/02/0602Clemmons.pdf.

[Cle10] Clements, P., and L. Bass, "The Business Goals Viewpoint," *IEEE Software*, vol. 27, no. 6, November–December 2010, pp. 38–45.

[CMM07] *Capability Maturity Model Integration (CMMI)*, Software Engineering Institute, 2007, available at www.sei.cmu.edu/cmmi/.

[CMM08] *People Capability Maturity Model Integration (People CMM)*, Software Engineering Institute, 2008, available at www.sei.cmu.edu/cmm-p/.

[Coa91] Coad, P., and E. Yourdon, *Object-Oriented Analysis*, 2nd ed., Prentice Hall, 1991.

[Coa99] Coad, P., E. Lefebvre, and J. DeLuca, *Java Modeling in Color with UML*, Prentice Hall, 1999.

[Coc01a] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer*, vol. 34, no. 11, November 2001, pp. 131–133.

[Coc01b] Cockburn, A., *Writing Effective Use-Cases*, Addison-Wesley, 2001.

[Coc02] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002.

[Coc04] Cockburn, A., "What the Agile Toolbox Contains," *CrossTalk*, November 2004, available at www.stsc.hill.af.mil/crosstalk/2004/11/0411Cockburn.html.

[Coc05] Cockburn, A., *Crystal Clear*, Addison-Wesley, 2005.

[Coc11] Cochran, C., *ISO 9001 in Plain English*, Paton Professional, 2011.

[Coh05] Cohn, M., "Estimating with Use Case Points," *Methods & Tools*, Fall, 2005, available at http://www.mountaingoatsoftware.com/articles/estimating-with-use-case-points.

[Col09] Collaris, R-A., and E. Dekker, "Software Cost Estimation Using Use Case Points, " IBM Developer Works, March 15, 2009, available at http://www.ibm.com/developerworks/rational/library/edge/09/mar09/collaris_dekker/.

[Con93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization, *CACM*, vol. 36, no. 10, October 1993, pp. 34–43.

[Con95] Constantine, L., "What DO Users Want? Engineering Usability in Software," *Windows Tech Journal*, December 1995, available from www.forUse.com.

[Con96] Conradi, R., "Software Process Improvement: Why We Need SPIQ," NTNU, October 1996, downloadable from www.idi.ntnu.no/grupper/su/publ/pdf/nik96-spiq.pdf.

[Con99] Constantine, L., and L. Lockwood, "Learning the Laws of Usability," *Software Development*, vol. 7, no. 10, October, 1999.

[Con02] Conradi, R., and A. Fuggetta, "Improving Software Process Improvement," *IEEE Software*, July–August 2002, pp. 2–9, available at http://citeseer.ist.psu.edu/conradi02improving.html.

[Cop05] Coplien, J., "Software Patterns," 2005, available at http://hillside.net/patterns/definition.html.

[Cou00] Coulouris, G., J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd ed., Addison-Wesley, 2000.

[Cri92] Christel, M., and K. Kang, "Issues in Requirements Elicitation," Software Engineering Institute, CMU/SEI-92-TR-12 7, September 1992.

[Crn11] Crnkovic, I., et al., "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, September–October 2011, pp. 593–615.

[Cro79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.

[Cur01] Curtis, B., W. Hefley, and S. Miller, *People Capability Maturity Model*, Addison-Wesley, 2001.

[Cur86] Curritt, P., M. Dyer, and H. Mills, "Certifying the Reliability of Software," *IEEE Trans, Software Engineering*, vol. SE-12, no. 1, January 1994.

[Cur90] Curtis, B., and D. Walz, "The Psychology of Programming in the Large: Team and Organizational Behavior," *Psychology of Programming*, Academic Press, 1990.

[CVS07] Concurrent Versions System, Ximbiot, 2007, available at http://ximbiot.com/cvs/wiki/index.php?title=Main_Page.

[CVS12] *Open CVS*, 2012, available at http://web.archive.org/web/20041220041434/http://www.opencvs.org/.

[DAC03] "An Overview of Model-Based Testing for Software," Data and Analysis Center for Software, CR/TA 12, June 2003, available from www.goldpractices.com/dwnload/practice/pdf/Model_Based_Testing.pdf.

[Dah72] Dahl, O., E. Dijkstra, and C. Hoare, *Structured Programming*, Academic Press, 1972.

[Dan09] Dang, A., "How Correct is Security by Correctness?" Tom's Hardware, July 16, 2009, available at http://www.tomshardware.com/reviews/joanna-rutkowska-rootkit,2356-7.html.

[Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, available at www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.

[Dar91] Dart, S., "Concepts in Configuration Management Systems," *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, available at www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.

[Dar99] Dart, S., "Change Management: Containing the Web Crisis," *Proc. Software Configuration Management Symposium*, Toulouse, France, 1999, available at www.perforce.com/perforce/conf99/dart.html.

[Dav90] Davenport, T. H., and J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review*, Summer 1990, pp. 11–27.

[Dav93] Davis, A., et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proceedings of the First International Software Metrics Symposium*, IEEE, Baltimore, MD, May 1993, pp. 141–152.

[Dav95a] Davis, M., "Process and Product: Dichotomy or Duality," *Software Engineering Notes*, ACM Press, vol. 20, no. 2, April 1995, pp. 17–18.

[Dav95b] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.

[Day99] Dayani-Fard, H., et al., "Legacy Software Systems: Issues, Progress, and Challenges," IBM Technical Report: TR-74.165-k, April 1999, available at www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.

[DeM02] DeMarco, T., and B. Boehm, "The Agile Methods, Fray," *IEEE Computer*, vol. 35, no. 6, June 2002, pp. 90–92.

[DeM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.

[DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.

[DeM98] DeMarco, T., and T. Lister, *Peopleware*, 2nd ed., Dorset House, 1998.

[Dem86] Deming, W., *Out of the Crisis*, MIT Press, 1986.

[Den73] Dennis, J., "Modularity," in *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Springer-Verlag, 1973, pp. 128–182.

[Des08] de Sá, M., and L. Carriço, "Lessons from Early Stages Design of Mobile Applications," *Proceedings of 10th International Conference on Human Computer Human with Mobile Services and Devices*, September 2008, pp. 127–136.

[deS09] de Sousa, C., et al, "Cooperative and Human Aspects of Software Engineering," *IEEE Software*, vol. 26, no. 6, pp. 17–19.

[Dev00] Devanbu, P., and S. Stubblebine, "Software Engineering for Security: A Roadmap," *Proc. ICSE*, IEEE, 2000, available at http://www0.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finaldevanbu.pdf.

[Dev01] Devedzik, V., "Software Patterns," in *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.

[Dha95] Dhama, H., "Quantitative Metrics for Cohesion and Coupling in Software," *Journal of Systems and Software*, vol. 29, no. 4, April 1995.

[Dij65] Dijkstra, E., "Programming Considered as a Human Activity," in *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.

[Dij72] Dijkstra, E., "The Humble Programmer," 1972 ACM Turing Award Lecture, *CACM*, vol. 15, no. 10, October 1972, pp. 859–866.

[Dij76a] Dijkstra, E., "Structured Programming," in *Software Engineering, Concepts and Techniques* (J. Buxton et al., ed.), Van Nostrand-Reinhold, 1976.

[Dij76b] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.

[Dij82] Dijksta, E., "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[Dix99] Dix, A., "Design of User Interfaces for the Web," *Proc. User Interfaces to Data Systems Conference*, September 1999, available at www.comp.lancs.ac.uk/computing/users/dixa/topics/webarch/.

[Don99] Donahue, G., S. Weinschenck, and J. Nowicki, "Usability Is Good Business," Compuware Corp., July 1999, available at www.compuware.com.

[Dre99] Dreilinger, S., "CVS Version Control for Web Site Projects," 1999, available at www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.

[Dru75] Drucker, P., *Management*, W. H. Heinemann, 1975.

[DSi08] D'Silva, V., et al., "Interview: Software Security in the Real World," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, July 2008, pp. 1165–1178.

[Duc01] Ducatel, K., et al., *Scenarios for Ambient Intelligence in 2010*, ISTAG-European Commission, 2001, downloadable from ftp://ftp.cordis.europa.eu/pub/ist/docs/istagscenarios2010.pdf.

[Dun01] Dunaway, D., and S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1,2 Method Description)*, Software Engineering Institute, 2001, available at www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.

[Dun02] Dunn, W., *Practical Design of Safety-Critical Computer Systems*, William Dunn, 2002.

[Dun82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.

[Duy02] VanDuyne, D., J. Landay, and J. Hong, *The Design of Sites*, Addison-Wesley, 2002.

[Dye92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

[Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends*, vol. 2, no. 5, May 1995, pp. 37–42.

[Eis01] Eisenstein, J., et al., "Applying Model-Based Techniques to the Development of UIs for Mobile Computers," *Proceedings of Intelligent User Interfaces*, January 2001.

[Eji91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.

[Elr01] Elrad, T., R. Filman, and A. Bader (ed.), "Aspect Oriented Programming," *Comm. ACM*, vol. 44, no. 10, October 2001, special issue.

[Erd09] Ergomus, H., "The Seven Traits of Superprofessionals," *IEEE Software*, vol. 26, no. 4, July–August, 2009, pp. 4–6.

[Erd10] Ergogmus, H., "Déjà vu: The Life of Software Engineering Ideas," *IEEE Software*, vol. 27, no. 1, January–February 2010, pp. 2–3.

[Eri05] Ericson, C., *Hazard Analysis Techniques for System Safety*, Wiley-Interscience, 2005.

[Eri08] Erickson, T., *The Interaction Design Patterns Page*, May 2008, available at www.visi .com/~snowfall/InteractionPatterns.html.

[Eva04] Evans, E., *Domain Driven Design*, Addison-Wesley, 2004.

[Eve09] Everett, G., and B. Meyer, "Point/Counterpoint," *IEEE Software*, vol. 26, no. 4, July–August 2009, pp. 62–65.

[Fag86] Fagan, M., "Advances in Software Inspections," *IEEE Trans. Software Engineering*, vol. 12, no. 6, July 1986.

[Fal10] Falessi, D., et al., "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 23–25.

[Fel07] Feller, J., et al. (eds.), *Perspectives on Free and Open Source Software*, The MIT Press, 2007.

[Fel89] Felican, L., and G. Zalateu, "Validating Halstead's Theory for Pascal Programs," *IEEE Trans. Software Engineering*, vol. SE-15, no. 2, December 1989, pp. 1630–1632.

[Fen91] Fenton, N., *Software Metrics*, Chapman and Hall, 1991.

[Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Engineering*, vol. SE-20, no. 3, March 1994, pp. 199–206.

[Fer00] Fernandez, E. B., and X. Yuan, "Semantic Analysis Patterns," *Proceedings of the 19th International Conerence on Conceptual Modeling, ER2000*, Lecture Notes in Computer Science 1920, Springer, 2000, pp. 183–195. Also available from www.cse.fau.edu/~ed/SAPpaper2.pdf.

[Fer97] Ferguson, P., et al., "Results of Applying the Personal Software Process," *IEEE Computer*, vol. 30, no. 5, May 1997, pp. 24–31.

[Fer98] Ferdinandi, P. L., "Facilitating Communication," *IEEE Software*, September 1998, pp. 92–96.

[Fil05] Filman, R., et al., *Aspect-oriented Software Development*, Addison-Wesley, 2005.

[Fir12] Firesmith, D., *Security and Safety Requirements for Software-Intensive Systems*, Auerbach, 2012.

[Fir93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.

[Fis11] Fisher, R., et al., *Getting to Yes: Negotiating without Giving In*, Penguin Books, 2011.

[Fle98] Fleming, Q., and J. Koppelman, "Earned Value Project Management," *CrossTalk*, vol. 11, no. 7, July 1998, p. 19.

[Fok10] Fokaefs, M., et al., WikiDev 2.0: Facilitating Software Development Teams, *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, March 15–18, 2010, pp. 276–277.

[Fos06] Foster, E., "Quality Culprits," InfoWorld Grip Line Weblog, May 2, 2006, available at http://weblog.infoworld.com/gripeline/2006/05/02_a395.html.

[Fow00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[Fow01] Fowler, M., and J. Highsmith, "The Agile Manifesto," *Software Development Magazine*, August 2001, available at www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.

[Fow02] Fowler, M., "The New Methodology," June 2002, available at www.martinfowler .com/articles/newMethodology.html#N8B.

[Fow03] Fowler, M., et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Fow04] Fowler, M., *UML Distilled*, 3rd ed., Addison-Wesley, 2004.

[Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[Fra93] Frankl, P., and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering*, vol. SE-19, no. 8, August 1993, pp. 770–787.

[Fre80] Freeman, P., "The Context of Design," in *Software Design Techniques*, 3rd ed. (P. Freeman and A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2–4.

[Fre90] Freedman, D., and G. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3rd ed., Dorset House, 1990.

[Fri10] Fricker, S., "Handshaking with Implementation Proposals: Negotiating Requirements Understanding," *IEEE Software*, vol. 27, no. 2, March–April 2010, pp. 72–80.

[Gag04] Gage, D., and J. McCormick, "We Did Nothing Wrong," *Baseline Magazine*, March 4, 2004, available at www.baselinemag.com/article2/0,1397,1544403,00.asp.

[Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences," Technical Report, University of Calgary, Calgary, Alberta, September 1995.

[Gam95] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gar08] GartnerGroup, "Understanding Hype Cycles," 2008, available at www.gartner.com/pages/story.php.id.8795.s.8.jsp.

[Gar09] Gardner, D., "Can Software Development Aspire to the Cloud?" *ZDNet.com*, April 28, 2009, available at http://www.zdnet.com/blog/gardner/can-software-development-aspire-to-the-cloud/2915.

[Gar09a] Garlan, D., et al., "Architectural Mismatch: Way Reuse is Still So Hard," *IEEE Software*, vol. 26, no. 4, July–August 2009, pp. 66–69.

[Gar10] Garcia-Crespo, A. et al., "A Qualitative Study of Hard Decision Making in Managing Global Software Development Teams," *Journal of Management Information Systems*, vol. 27, no. 3, June 2010, pp. 247–252.

[Gar12] GartnerGroup, "Gartner Says Worldwide Media Tablets Sales to Reach 119 Million Units in 2012," April 2012, available at http://www.gartner.com/it/page.jsp?id=1980115.

[Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review*, Fall 1984, pp. 25–45.

[Gar87] Garvin D., "Competing on the Eight Dimensions of Quality," *Harvard Business Review*, November 1987, pp. 101–109. A summary is available at www.acm.org/crossroads/xrds6-4/software.html.

[Gar95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.

[Gau89] Gause, D., and G. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[Gav11] Gavalas, D., and D. Economou, "Development Platforms for Mobile Applications," *IEEE Software*, vol. 28, no. 1, January–February, 2011, pp. 77–86.

[Gey01] Geyer-Schulz, A., and M. Hahsler, "Software Engineering with Analysis Patterns," Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wirschaftsuniversität Wien, November 2001, downloadable from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.8415&rep=rep1&type=pdf.

[Gho01] Ghoshm, A., and Swaminatha, T. "Software Security and Privacy Risks in Mobile E-Commerce," *Communication of the ACM*, vol. 44, no. 2, February 2001, pp. 51–57.

[Gil06] Gillis, D., "Pattern-Based Design," tehan + lax blog, September 14, 2006, available at www.teehanlax.com/blog/?p=96.

[Gil88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.

[Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, January 1995.

[Gla02] Gladwell, M., *The Tipping Point*, Back Bay Books, 2002.

[Gla98] Glass, R., "Defining Quality Intuitively," *IEEE Software*, May 1998, pp. 103–104, 107.

[Gli07] Glinz, M., and R. Wieringa, "Stakeholders in Requirements Engineering," *IEEE Software*, vol. 24, no. 2, March–April 2007, pp. 18–20.

[Gli09] Glinz, M., "A Risk-Base, Value-Oriented Approach to Quality Requirements," *IEEE Software*, vol. 26, no. 5, March–April 2009, pp. 34–41.

[Glu94] Gluch, D., "A Construct for Describing Software Development Risks," CMU/SEI-94-TR-14, Software Engineering Institute, 1994.

[Gna99] Gnaho, C., and F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering," *Proceedings of the First ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999.

[Gon04] Gonzales, R., "Requirements Engineering," Sandia National Laboratories, a slide presentation, available at www.incose.org/enchantment/docs/04AprRequirementsEngineering.pdf.

[Gor06] Gorton, I., *Essential Software Architecture*, Springer, 2006.

[Got11] Gotel, O., and S. Morris, "Requirements Tracery," *IEEE Software*, vol. 28, no. 5, September–October 2011, pp. 92–94.

[Gra03] Gradecki, J., and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.

[Gra87] Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.

[Gra92] Grady, R. G., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.

[Gru00] Gruia-Catalin, R., et al., "Software Engineering for Mobility: A Roadmap," *Proceedings of the 22nd International Conference on the Future of Software Engineering*, 2000.

[Gru02] Grundy, J., "Aspect-Oriented Component Engineering," 2002, www.cs.auckland.ac.nz/~john-g/aspects.html.

[Gub09] Gube, J., "40+ Helpful Resources on User Interface Design Patterns," June 15, 2009, http://www.smashingmagazine.com/2009/06/15/40-helpful-resources-on-user-interface-design-patterns/.

[Gus89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment," *Proceedings of the Second International Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 114–117.

[Gut93] Guttag, J., and J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Hac98] Hackos, J., and J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.

[Hai02] Hailpern, B., and P. Santhanam, "Software Debugging, Testing and Verification," *IBM Systems Journal*, vol. 41, no. 1, 2002, available at www.research.ibm.com/journal/sj/411/hailpern.html.

[Hal77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.

[Hal90] Hall, A., "Seven Myths of Formal Methods," *IEEE Software*, September 1990, pp. 11–20.

[Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.

[Ham90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate," *Harvard Business Review*, July–August 1990, pp. 104–112.

[Han95] Hanna, M., "Farewell to Waterfalls," *Software Magazine*, May 1995, pp. 38–46.

[Har11] Harris, N., and P. Avgeriou, "Pattern-Based Architecture Reviews," *IEEE Software*, vol. 28, no. 6, November–December 2011, pp. 66–71.

[Har12] Hardy, T., *Software and System Safety*, Authorhouse, 2012.

[Har98b] Harrison, R., S. Counsell, and R. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Engineering*, vol. SE-24, no. 6, June 1998, pp. 491–496.

[Hen10] Hendler, J., "Web 3.0: The Dawn of Semantic Search," *IEEE Computer*, January, 2010, p. 77.

[Her00] Herrmann, D., *Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000.

[Het84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.

[Het93] Hetzel, W., *Making Software Measurement Work*, QED Publishing, 1993.

[Hev93] Hevner, A., and H. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems Journal*, vol. 31, no. 2, February 1993, pp. 232–251.

[Hig01] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 1," *Cutter IT Journal.*, vol. 14, no. 12, December 2001.

[Hig02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2," *Cutter IT Journal*, vol. 15, no. 1, January 2002.

[Hig95] Higuera, R., "Team Risk Management," *CrossTalk*, U.S. Dept. of Defense, January 1995, pp. 2–4.

[Hil05] Hildreth, S., "Buggy Software: Up from a Low Quality Quagmire," *Computerworld*, July 25, 2005, available at www.computerworld.com/developmenttopics/development/story/0,10801,103378,00.html.

[Hil13] Hillside.net, *Patterns Catalog*, 2013, available at http://hillside.net/patterns/patterns-catalog.

[Hne11] Hneif, M., and S. Lee, "Using Guidelines to Improve Quality in Software Nonfunctional Attributes," *IEEE Software*, vol. 28, no. 5, November–December 2011, pp. 72–73.

[Hof00] Hofmeister, C., R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.

[Hof01] Hofmann, C., et al., "Approaches to Software Architecture," 2001, downloadable from http://citeseer.nj.nec.com/84015.html.

[Hol06] Holzner, S., *Design Patterns for Dummies*, For Dummies Publishers, 2006.

[Hoo96] Hooker, D., "Seven Principles of Software Development," September 1996, available at http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.

[Hoo12] Hoober, S., and E. Berkman, *Designing Mobile Interfaces*, O'Reilly Media, 2012.

[Hop90] Hopper, M., "Rattling SABRE, New Ways to Compete on Information," *Harvard Business Review*, May–June 1990.

[Hor03] Horch, J., *Practical Guide to Software Quality Management*, 2nd ed., Artech House, 2003.

[Hoy09] Hoyle, D., *ISO 9000 Quality Systems Handbook*, 6th ed., Taylor & Francis, 2009.

[Hum00] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, 2000.

[Hum05] Humphrey, W., *A Self-Improvement Process for Software Engineers*, Addison-Wesley, 2005.

[Hum95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[Hum96] Humphrey, W., "Using a Defined and Measured Personal Software Process," *IEEE Software*, vol. 13, no. 3, May–June 1996, pp. 77–88.

[Hum97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

[Hum98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process," *CrossTalk*, April 1998, available at www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp.

[Hun99] Hunt, A., D. Thomas, and W. Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 1999.

[Hya96] Hyatt, L., and L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," NASA SATC, 1996, available at http://satc.gsfc.nasa.gov/support/STC_APR96/qualtiy/stc_qual.html.

[IBM13] IBM, *Web Services Globalization Model*, 2013, available at http://www.ibm.com/developerworks/webservices/library/ws-global/.

[IBM81] "Implementing Software Inspections," course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.

[IEE93a] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.

[IEE93b] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993.

[IEE00] IEEE Standard Association, IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000, available at http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html.

[IEE05] IEEE Std. 982.1-2005, *IEEE Standard Dictionary of Measures of the Software Aspects of Dependability*, 2005, available at http://standards.ieee.org/findstds/standard/982.1-2005.html.

[IEE09] *IEEE Software* (special issue), "Human Aspects of Software Engineering," vol. 28, no 6, November–December, 2009.

[IFP01] *Function Point Counting Practices Manual*, 2001, downloadable from: perun.pmf.uns.ac.rs/old/repository/.../se/functionpoints.pdf.

[IFP12] *Function Point Bibliography/Reference Library*, International Function Point Users Group, 2012, available from http://www.ifpug.org/?page_id=237.

[Isk08] Iskold, A., "Top Ten Concepts that Every Software Engineer Should Know," ReadWrite, July 2008, available from http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know.

[ISO00] *ISO 9001: 2000 Document Set*, International Organization for Standards, 2000, available at www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.

[ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.

[ISO08] ISO SPICE, 2008, available at www.isospice.com/categories/SPICE-Project/.

[Ivo01] Ivory, M., R. Sinha, and M. Hearst, " Empirically Validated Web Page Design Metrics," ACM *SIGCHI'01*, March 31–April 4, 2001, available at http://webtango.berkeley.edu/papers/chi2001/.

[Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal*, vol. 15, no. 1, January 2002, pp. 18–24.

[Jac02b] Jacyntho, D., D. Schwabe, and G. Rossi, "An Architecture for Structuring Complex Web Applications," 2002, available at www2002.org/CDROM/alternate/478/.

[Jac04] Jacobson, I., and P. Ng, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Jac75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.

[Jac92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

[Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software*, July 1998, pp. 43–45.

[Jac99] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jal04] Jalote, P., et al., "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software*, vol. 70, issue 2, 2004, pp. 117–127. Available at www.cse.iitk.ac.in/users/jalote/papers/Timeboxing.pdf.

[Jec06] Jech, T., *Set Theory*, 3rd ed., Springer, 2006.

[Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success," *CrossTalk*, October 2004. Available at www.stsc.hill.af.mil/crossTalk/2004/10/0410Jones.html.

[Jon86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.

[Jon91] Jones, C., *Systematic Software Development Using VDM*, 2nd ed., Prentice Hall, 1991.

[Jon96] Jones, C., "How Software Estimation Tools Work," *American Programmer*, vol. 9, no. 7, July 1996, pp. 19–27.

[Jon98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.

[Joy00] Joy, B., "The Future Doesn't Need Us," *Wired*, vol. 8, no. 4, April 2000.

[Kai02] Kaiser, J., "Elements of Effective Web Design," *About, Inc.*, 2002, available at http://webdesign.about.com/library/weekly/aa091998.htm.

[Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, available at www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.

[Kan93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed., Van Nostrand-Reinhold, 1993.

[Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs," 1995, available at www.badsoftware.com/plaintif.htm.

[Kanll] Kannan, N., "Mobile Testing: Nine Strategy Tests You'll Want to Perform," 2011, available at http://searchsoftwarequality.techtarget.com/tip/Mobile-testing-Nine-strategy-tests-youll-want-to-perform.

[Kar94] Karten, N., *Managing Expectations*, Dorset House, 1994.

[Kau95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.

[Kaz03] Kazman, R., and A. Eden, "Defining the Terms Architecture, Design, and Implementation," *news@sei interactive*, Software Engineering Institute, vol. 6, no. 1, 2003, available at www.sei.cmu.edu/news-at-sei/columns/the_architect/2003/1q03/architect-1q03.htm.

[Kaz98] Kazman, R., et al., *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, July 1998, summarized at http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm.

[Kea07] Keane, "Testing Mobile Business Applications," a white paper, 2007, available at www.keane.com.

[Kei98] Keil, M., et al., "A Framework for Identifying Software Project Risks," *CACM*, vol. 41, no. 11, November 1998, pp. 76–83.

[Kel00] Kelly, D., and R. Oshana, "Improving Software Quality Using Statistical Techniques, Information and Software Technology," *Elsevier*, vol. 42, August 2000, pp. 801–807, available at www.eng.auburn.edu/~kchang/comp6710/readings/Improving_Quality_with_Statistical_Testing_InfoSoftTech_August2000.pdf.

[Ker05] Kerievsky, J., *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium, Executive Report, vol. 6., no. 2, 2005, available at www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/apmr0502.html.

[Ker11] Kershbaum, F., et al., "Secure Collaborative Supply-Chain Management," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 38–43.

[Ker78] Kernighan, B., and P. Plauger, *The Elements of Programming Style*, 2nd ed., McGraw-Hill, 1978.

[Kho12a] Khode, A., "Getting Started with Mobile Apps Testing," 2012, available at http://www.mobileappstesting.com/getting-started-with-mobile-apps-testing/.

[Kho12b] Khode, A., "Checklist for Mobile Test Automation Tools," 2012, available at http://www.mobileappstesting.com/getting-started-with-mobile-apps-testing/

[Kir94] Kirani, S., and W. Tsai, "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.

[Kiz05] Kizza, J., *Computer Network Security*, Springer, 2005.

[Knu98] Knuth, D., *The Art of Computer Programming*, three volumes, Addison-Wesley, 1998.

[Koe12] Koester, J., "The Seven Deadly Sins of MobileApp Design," *Venture Beat/Mobile*, May31,2012,availableathttp://venturebeat.com/2012/05/31/the-7-deadly-sins-of-mobile-app-design/

[Kon02] Konrad, S., and B. Cheng, "Requirements Patterns for Embedded Systems," *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, IEEE, September 2002, pp. 127–136, available at http://citeseer.ist.psu.edu/669258.html.

[Kor03] Korpipaa, P., et al., "Managing Context Information in Mobile Devices," *IEEE Pervasive Computing*, vol. 2, no. 3, July–September 2003, pp. 42–51.

[Kra88] Krasner, G., and S. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, August–September 1988, pp. 26–49.

[Kra95] Kraul, R., and L. Streeter, "Coordination in Software Development," *CACM*, vol. 38, no. 3, March 1995, pp. 69–81.

[Kru05] Krutchen, P., "Software Design in a Postmodern Era," *IEEE Software*, vol. 22, no. 2, March–April 2005, pp. 16–18.

[Kru06] Kruchten, P., H. Obbink, and J. Stafford (eds.), "Software Architectural" (special issue), *IEEE Software*, vol. 23, no. 2, March–April 2006.

[Kru09] Kruchten, P., et al. "The Decision View's Role in Software Architecture Practice," *IEEE Software*, vol. 26, no. 2, March–April 2009, pp. 70–72.

[Kur05] Kurzweil, R., *The Singularity Is Near*, Penguin Books, 2005.

[Kur13] Kurzweil, R., *How to Create a Mind*, Viking, 2013.

[Kyb84] Kyburg, H., *Theory and Measurement*, Cambridge University Press, 1984.

[Laa00] Laakso, S., et al., "Improved Scroll Bars," *CHI 2000 Conf. Proc.*, ACM, 2000, pp. 97–98, available at www.cs.helsinki.fi/u/salaakso/patterns/.

[Lag10] Lago, P., et al., "Software Architecture: Framing Stakeholders' Concerns," *IEEE Software*, vol. 27, no. 6, November–December 2010, pp. 20–24.

[Lai02] Laitenberger, A., "A Survey of Software Inspection Technologies," in *Handbook on Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2002.

[Lam01a] Lam, W., "Testing E-Commerce Systems: A Practical Guide," *IEEE IT Pro*, March–April 2001, pp. 19–28.

[Lam01b] Lamsweerde, A. "Goal-Oriented Requirements Engineering: A Guided Tour," *Proceedings of 5th IEEE International Symposium on Requirements Engineering*, Toronto, August 2009, pp. 249–263.

[Lan01] Lange, M., "It's Testing Time! Patterns for Testing Software," June 2001, available at www.testing.com/test-patterns/patterns/index.html.

[Lan02] Land, R., "A Brief Survey of Software Architecture," technical report, Dept. of Computer Engineering, Mälardalen University, Sweden, February 2002.

[Lan10] Lanubile, F., C. Ebert, R. Prikladnicki, and A. Vizcaino, "Collaboration Tools for Global Software Engineering," *IEEE Software*, vol. 27, 2010, pp. 52–55.

[Laz11] Lazzaroni, M., et al., *Reliability Engineering*, Springer, 2011.

[Leh97a] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.

[Leh97b] Lehman, M., et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, available at www.ece.utexas.edu/~perry/work/papers/feast1.pdf.

[Let01] Lethbridge, T., and R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.

[Let03a] Lethbridge, T., Personal communication on domain analysis, May 2003.

[Let03b] Lethbridge, T., Personal communication on software metrics, June 2003.

[Lev01] Levinson, M., "Let's Stop Wasting $78 billion a Year," *CIO Magazine*, October 15, 2001, available at www.cio.com/archive/101501/wasting.html.

[Lev95] Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[Lew09] Lewicki, R., B. Barry, and D. Saunders, *Negotiation*, McGraw-Hill, 2009.

[Lie03] Lieberherr, K., "Demeter: Aspect-Oriented Programming," May 2003, available at www.ccs.neu.edu/home/lieber/LoD.html.

[Lin79] Linger, R., H. Mills, and B. Witt, *Structured Programming*, Addison-Wesley, 1979.

[Lin88] Linger, R., and H. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. COMPSAC '88*, Chicago, October 1988.

[Lin94] Linger, R., "Cleanroom Process Model," *IEEE Software*, vol. 11, no. 2, March 1994, pp. 50–58.

[Lip10] Lipner, S., "The Security Development Life Cycle," June 24, 2010, available at https://www.owasp.org/images/7/78/OWASP_AppSec_Research_2010_Keynote_2_by_Lipner.pdf.

[Lis88] Liskov, B., "Data Abstraction and Hierarchy," *SIGPLAN Notices*, vol. 23, no. 5, May 1988.

[Liu98] Liu, K., et al., "Report on the First SEBPC Workshop on Legacy Systems," Durham University, February 1998, available at www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.

[Lon02] Longstreet, D., "Fundamental of Function Point Analysis," Longstreet Consulting, Inc., 2002, available at www.ifpug.com/fpafund.htm.

[Lor94] Lorenz, M., and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.

[Lup08] Lupton, E., and J. Cole, *Graphic Design: The New Basics*, Princeton Architectural Press, 2008.

[Maa07] Maassen, O., and S. Stelting, "Creational Patterns: Creating Objects in an OO System," 2007, available at www.informit.com/articles/article.asp?p=26452&rl=1.

[Mad10] Madison, J., "Agile-Architecture Interactions," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 41–48.

[Mai10a] Maiden, N., and S. Jones, "Agile Requirements—Can We Have Our Cake and Eat It Too?" *IEEE Software*, vol. 27, no. 3, May–June 2010, pp. 20–24.

[Mai10b] Maiden, N., "Service Design: It's All in the Brand," *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 18–19.

[Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM*, vol. 24, no. 3, March 1981, pp. 106–113.

[Man97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.

[Mar00] Martin, R., "Design Principles and Design Patterns," 2000, available at www.objectmentor.com.

[Mar01] Marciniak, J. (ed.), *Encyclopedia of Software Engineering*, 2nd ed., Wiley, 2001.

[Mar02a] Marick, B., "Software Testing Patterns," 2002, available at www.testing.com/test-patterns/index.html.

[Mar94] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1994.

[Mas02] Mascolo, C., et al., "Mobile Computing Middleware," appears in *Advanced Lectures on Networking* (E. Georgi, ed.), Springer-Verlag, 2002.

[Mat94] Matson, J., et al, "Software Cost Estimation Using Function Points," IEEE Trans. Software Engineering, vol. SE-20, no. 4, April 1994, pp. 275–287.

[McC04] McConnell, S., *Code Complete*, 2nd. ed., Microsoft Press, 2004.

[McC09] McCaffrey, J., "Analyzing Risk Exposure and Risk using PERIL," *MSDN Magazine*, January 2009, available at http://msdn.microsoft.com/en-us/magazine/dd315417.aspx

[McC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308–320.

[McC77] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977.

[McC94] McCabe, T. J., and A. H. Watson, "Software Complexity," *CrossTalk*, vol. 7, no. 12, December 1994, pp. 5–9.

[McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 143–144.

[McC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.

[McC99] McConnell, S., "Software Engineering Principles," *IEEE Software*, vol. 16, no. 2, March–April 1999, available at www.stevemcconnell.com/ieeesoftware/eic04.htm.

[McD93] McDermid, J., and P. Rook, "Software Development Process Models," in *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26–15/28.

[McG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes*, vol. 16, no. 4, October 1991, pp. 53–54.

[McG94] McGregor, J., and T. Korson, "Integrated Object-Oriented Testing and Development Processes," *Communications of the ACM*, vol. 37, no. 9, September, 1994, pp. 59–77.

[McN10] McNeil, P., *The Web Designer's Idea Book*, vol. 2, How Publishers, 2010.

[Mea10] Mead, N., and Jarzombek, J., "Advancing Software Assurance with Public-Private Collaboration," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 21–30.

[Mei06] Meier, J., "Web Application Security Engineering," *IEEE Security and Privacy*, July–August 2006, pp. 16–24.

[Mei09] Meier, J., et al., *Microsoft Application Architecture Guide*, 2nd ed., Microsoft Press, 2009, available at http://msdn.microsoft.com/en-us/library/ff650706.

[Mei12] Meier, J., et al., "Chapter 19: Mobile Applications," *Application Architecture Guide, 2.0*, 2012, available at http://apparchguide.codeplex.com/wikipage?title=Chapter%2019%20-%20Mobile%20Applications

[Men01] Mendes, E., N. Mosley, and S. Counsell, "Estimating Design and Authoring Effort," *IEEE Multimedia*, vol. 8, no. 1, January–March 2001, pp. 50–57.

[Mer93] Merlo, E., et al., "Reengineering User Interfaces," *IEEE Software*, January 1993, pp. 64–73.

[Mes08] Messeguer, R., et al. "Communication and Coordination Patterns to Support Mobile Collaboration," *Proceedings of 12th International Conference on Collaborative and Cooperative Work*, April 2008, pp. 565–570.

[Mey09] Meyer, B., et al., "Programs that Test Themselves," *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 46–55.

[Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns," *MSDN*, May 2004, available at http://msdn2.microsoft.com/en-us/library/ms978700.aspx.

[Mic12] "Principles of Service-Oriented Design," Microsoft, 2012, available at http://msdn.microsoft.com/en-us/library/bb972954.aspx

[Mic13a] Microsoft, "Patterns and Practices," *MSDN*, available at http://msdn.microsoft.com/en-us/library/ff647589.aspx.

[Mic13b] *Microsoft Accessibility Technology for Everyone*, 2013, available at www.microsoft.com/enable/.

[Mil00a] Miller, E., "WebSite Testing," 2000, available at www.soft.com/eValid/Technology/White.Papers/website.testing.html.

[Mil04] Miler, J., and J. Gorski, "Risk Identification Patterns for Software Projects," *Foundations of Computing and Decision Sciences*, vol. 29, no. 1, 2004, pp. 115–131, available at http://iag.pg.gda.pl/iag/download/Miler-Gorski_Risk_Identification_Patterns.pdf.

[Mil72] Mills, H., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.

[Mil77] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1–3.

[Mil87] Mills, H., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, September 1987, pp. 19–25.

[Mil88] Mills, H., "Stepwise Refinement and Verification in Box Structured Systems," *Computer*, vol. 21, no. 6, June 1988, pp. 23–35.

[Min95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.

[Mob11] Mobile Labs, "Mobile Application Test Automation: Best Practices for Best Results," a white paper, 2011, available at http://mobilelabsinc.com/wp-content/uploads/2012/01/Mobile-Application-Test-Automation-Best-Practices-White-Paper.pdf.

[Mob12] "Mobile UI Patterns," 2012, available at http://mobile-patterns.com/.

[Mor05] Morales, A., "The Dream Team," *Dr. Dobbs Portal*, March 3, 2005, available at www.ddj.com/dept/global/184415303.

[Mor81] Moran, T., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3–50.

[Mus87] Musa, J., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[Mye78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.

[Mye79] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[Myl11] Malavarapu, V., and M. Inanamdar, "Taking Testing to the Cloud," 2012, available at http://www.cognizant.com/InsightsWhitepapers/Taking-Testing-to-the-Cloud.pdf.

[NAS07] NASA, *Software Risk Checklist*, Form LeR-F0510.051, March 2007, available at http://osat-ext.grc.nasa.gov/rmo/spa/SoftwareRiskChecklist.doc.

[Nei11] Neil, T., "A Look Inside Mobile Design Patterns," UXBooth.com, November 12, 2011, available at http://www.uxbooth.com/blog/mobile-design-patters/.

[Nei12] Neil, T., *Mobile Design Pattern Gallery*, O'Reilly Media, 2012.

[Ngu00] Nguyen, H., "Testing Web-Based Applications," *Software Testing and Quality Engineering*, May–June 2000, available at www.stqemagazine.com.

[Ngu01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.

[Nie00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.

[Nie92] Nierstrasz, O., S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *CACM*, vol. 35, no. 9, September 1992, pp. 160–165.

[Nie94] Nielsen, J., and J. Levy, "Measuring Usability: Preference vs. Performance," *CACM*, vol. 37, no. 4, April 1994, pp. 65–75.

[Nie96] Nielsen, J., and A. Wagner, "User Interface Design for the WWW," *Proc. CHI '96 Conf. on Human Factors in Computing Systems*, ACM Press, 1996, pp. 330–331.

[Nir10] Niranjan, P., and C. V. Guru Rao, "A Mockup Tool for Software Component Reuse Library," *Intl. J. Software Engineering & Applications*, vol. 1, no. 2, April 2010, available at http://airccse.org/journal/ijsea/papers/0410ijsea1.pdf.

[Nok13] "Category: Mobile Design," Nokia Developer, 2013, available at http://www.developer.nokia.com/Community/Wiki/Category:Mobile_Design_Patterns.

[Nog00] Nogueira, J., C. Jones, and Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering," Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, June 2000, available at www.dodccrp.org/2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.

[Nor70] Norden, P., "Useful Tools for Project Management" in *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.

[Nor86] Norman, D. A., "Cognitive Engineering," in *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.

[Nor88] Norman, D., *The Design of Everyday Things*, Doubleday, 1988.

[Nov04] Novotny, O., "Next Generation Tools for Object-Oriented Development," *The Architecture Journal*, January 2005, available at http://msdn2.microsoft.com/en-us/library/aa480062.aspx.

[Noy02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an online publication of Fawcette Technical Publications), June 2002, available at www.fawcette.com/resources/managingdev/methodologies/scrum/.

[Nun11] Nunes, N., L. Constantine, and R. Kazman, "iUCP: Estimating Interactive Software Project Size with Enhanced Use Case Points," *IEEE Software*, vol. 28, no. 4, July–August 2011, pp. 64–73.

[Obj10] "The Dependency Inversion Principle," *Objectmentor.com*, 2010, available at www.objectmentor.com/resources/articles/dip.pdf>.

[Off02] Offutt, J., "Quality Attributes of Web Software Applications," *IEEE Software*, March–April 2002, pp. 25–32.

[Ols06] Olsen, G., "From COM to Common," *Component Technologies*, ACM, vol. 4, no. 5, June 2006, available at http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=394.

[Ols99] Olsina, L., et al., "Specifying Quality Characteristics and Attributes for Web Sites," *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999.

[OMG03a] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.5, March 2003, available at www.rational.com/uml/resources/documentation/.

[OMG03b] "Object Constraint Language Specification," in *Unified Modeling Language*, v2.0, Object Management Group, September 2003, available at www.omg.org.

[Orf99] Orfali, R., D. Harkey, and J. Edwards, *Client/Server Survival Guide*, 3rd ed., Wiley, 1999.

[Osb90] Osborne, W. M., and E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle," *IEEE Software*, January 1990, pp. 10–11.

[OSO12] *OpenSource.org*, 2012, available at www.opensource.org/.

[Pag85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.

[Par72] Parnas, D., "On Criteria to Be Used in Decomposing Systems into Modules," *CACM*, vol. 14, no. 1, April 1972, pp. 221–227.

[Par96a] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.

[Par96b] Park, R. E., W. B. Goethert, and W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.

[Par10] Parnas, D., "Interview: Software Security in the Real World," *IEEE Computer*, vol. 43, no. 1, January 2010, pp. 28–34.

[Par11] Pardo, C., et al., "Harmonizing Quality Assurance Processes and Product Characteristics," *IEEE Computer*, June 2011, pp. 94–96.

[Pas10] Passos, L., et al., "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 82–89,

[Pat07] Patton, J., "Understanding User Centricity," *IEEE Software*, vol. 24, no. 6, November–December 2007, pp. 9–11.

[Pau94] Paulish, D., and A. Carleton, "Case Studies of Software Process Improvement Measurement," *Computer*, vol. 27, no. 9, September 1994, pp. 50–57.

[Pea11] Pearson, S., and M. Mont, "Sticky Policies: An Approach for Managing Privacy across Multiple Parties," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 60–68.

[Pee11] Peeters, J., "Agile Security Requirements Engineering," *Proceedings of First International Workshop on Empirical Requirements Engineering*, 2011.

[Per74] Persig, R., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, 1974.

[Pha89] Phadke, M., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.

[Pha97] Phadke, M., "Planning Efficient Software Tests," *CrossTalk*, vol. 10, no. 10, October 1997, pp. 11–15.

[Phi02] Phillips, M., "CMMI V1.1 Tutorial," April 2002, available at www.sei.cmu.edu/cmmi/.

[Phi98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.

[Pol45] Polya, G., *How to Solve It*, Princeton University Press, 1945.

[Poo88] Poore, J., and H. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress*, November 1988, pp. 52–55.

[Poo93] Poore, J., H. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software*, vol. 10, no. 1, January 1993, pp. 88–99.

[Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve*, 2008, available at www.faithpopcorn.com/.

[Pot04] Potter, M., *Set Theory and Its Philosophy: A Critical Introduction*, Oxford University Press, 2004.

[Pow02] Powell, T., *Web Design*, 2nd ed., McGraw-Hill/Osborne, 2002.

[Pow98] Powell, T., *Web Site Engineering*, Prentice Hall, 1998.

[Pra07] Pratt, M., "Five Tips for Building an Incident Response Plan," *Computerworld*, May 16, 2007, available at http://www.computerworld.com/s/article/9019558/Five_tips_for_building_an_incident_response_pla.

[Pre05] Pressman, R., *Adaptable Process Model*, Version 2.0, R. S. Pressman & Associates, 2005, available at www.rspa.com/apm/index.html.

[Pre08] Pressman, R., and D. Lowe, *Web Engineering: A Practitioner's Approach*, McGraw-Hill, 2008.

[Pre88] Pressman, R., *Making Software Engineering Happen*, Prentice Hall, 1988.

[Pre94] Premerlani, W., and M. Blaha, "An Approach for Reverse Engineering of Relational Databases," *CACM*, vol. 37, no. 5, May 1994, pp. 42–49.

[Pri10] Prince, B., "10 Most Dangerous Web App Security Flaws," *eWeek.com*, April, 19, 2010, available at http://www.eweek.com/c/a/Security/10-Most-Dangerous-Web-App-Security-Risks-730757/.

[Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Trans. Software Engineering*, vol. SE-4, no. 4, July 1978, pp. 345–361.

[Put92] Putnam, L., and W. Myers, *Measures for Excellence*, Yourdon Press, 1992.

[Put97a] Putnam, L., and W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software*, November 1997, pp. 105–107.

[Put97b] Putnam, L., and W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.

[Pyz03] Pyzdek, T., *The Six Sigma Handbook*, McGraw-Hill, 2003.

[QAI08] *A Software Engineering Curriculum*, QAI, 2008, information available at www.qaie-school.com/innerpages/offer.asp.

[QSM02] "QSM Function Point Language Gearing Factors," Version 2.0, *Quantitative Software Management*, 2002, available at www.qsm.com/FPGearing.html.

[Qur09] Qureshi N., and A. Perini, "Engineering Adaptive Requirements," *Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Vancouver, May 2009, pp. 126–131.

[Rad02] Radice, R., *High-Quality Low Cost Software Inspections*, Paradoxicon Publishing, 2002.

[Rai06] Raiffa, H., *The Art and Science of Negotiation*, Belknap Press, 2005.

[Rat12] Raatikainen, M., et al., "Mobile Content as a Service: A Blueprint for a Vendor-Neutral Cloud of Mobile Devices," *IEEE Software*, vol. 29, no. 4, July–August, 2012, pp. 28–32.

[Red10] Redwine, S., "Fitting Software Assurance into Higher Education," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 41–66.

[Ree99] Reel, J., "Critical Success Factors in Software Projects," *IEEE Software*, May 1999, pp. 18–23.

[Reu12] Reuveni, D. "Crowdsourcing Provides Answer to App Testing Dilemma," 2012, available at http://www.wirelessweek.com/Articles/2010/02/Mobile-Content-Crowdsourcing-Answer-App-Testing-Dilemma-Mobile-Applications/.

[Ric01] Ricadel, A., "The State of Software Quality," *InformationWeek*, May 21, 2001, available at www.informationweek.com/838/quality.htm.

[Ric04] Rico, D., *ROI of Software Process Improvement*, J. Ross Publishing, 2004. A summary article can be found at http://davidfrico.com/rico03a.pdf.

[Rob10] Robinson, W., "A Roadmap for Comprehensive Requirements Monitoring," *IEEE Computer*, vol. 43, no. 5, May 2010, pp. 64–72.

[Roc06] *Graphic Design That Works*, Rockport Publishers, 2006.

[Roc11] Rocha, F., S, Abreus, and M. Correia, "The Final Frontier: Confidentiality and Privacy in the Cloud," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 44–50.

[Roc94] Roche, J. M., "Software Metrics and Measurement Principles," *Software Engineering Notes*, ACM, vol. 19, no. 1, January 1994, pp. 76–85.

[Rod12] Rodriguez, S., "Half of Americans Now Have Smartphones," *The Los Angeles Times*, August 14, 2012, available at http://www.latimes.com/business/technology/la-fi-tn-americans-smartphones-20120814,0,6077673.story.

[Rod98] Rodden, T., et al., "Exploiting Context in HCI Design for Mobile Systems," *Proceedings of Workshop on Human Computer Interaction with Mobile Devices*, 1998.

[Roe00] Roetzheim, W., "Estimating Internet Development," *Software Development*, August 2000, available at www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm.

[Rog12] Rogers, A., "Software Quality Assurance Engineers Are the Happiest Workers in America," *Business Insider*, April 16, 2012, available at http://www.businessinsider.com/happiest-jobs-in-america-2012-4.

[Rom11] Roman, R., et al., "Securing the Internet of Things," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 51–58.

[Roo09] Rooksby, J., et al., "Testing in the Wild: The Social and Organizational Dimensions of Real World Practice," *Journal of Computer Supported Work*, vol. 18, no. 5–6, December 2009, pp. 559–580.

[Roo96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes," 1996, available at www.imd.ch/fac/roos/paper_po.html.

[Ros04] Rosenhainer, L., "Identifying Crosscutting Concerns in Requirements Specifications," 2004, available at http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/Rosenhainer.pdf.

[Ros75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer*, vol. 8, no. 5, May 1975.

[Rot02] Roth, J., "Seven Challenges for Developers of Mobile Groupware," in *Proceedings of Computer Human Interaction Workshop on Mobile Ad Hoc Collaboration*, 2002.

[Rou02] Rout, T. (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide*, 2002, available at www.sqi.gu.edu.au/spice/suite/download.html.

[Roy70] Royce, W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. WESCON*, August 1970.

[Roz11] Rozanski, N., and E. Woods, *Software Systems Architecture*, second edition, Addison-Wesley, 2011.

[Rum91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Rya11a] Ryan, M., "Cloud Computing Privacy Concerns on Our Doorstep," *Communication of the ACM*, vol. 44, no. 2, January 2011, pp. 36–38.

[Sae11] Saeed, A., T. Chen, and O. Alzubi, "Malicious and Spam Posts in Online Social Networks," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 23–28.

[Saf08] Safanov, V., *Using Aspect-Oriented Programming for Trustworthy Software Development*, Wiley-Interscience, 2008.

[Sai11] Saieddian, H., and D. Broyles, "Security Vulnerabilities in the Same Origin Policy Implications and Alternatives," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 29–36.

[Sal09] Saleh, K., and G. Elshahry, "Modeling Security Requirements for Trustworthy Systems," *Encyclopedia of Information Science and Technology*, 2nd ed., 2009.

[Sar06] Sarwate, A., "Hot or Not: Web Application Vulnerabilities," *SC Magazine*, December 27, 2006, available at http://www.scmagazine.com/hot-or-not-web-application-vulnerabilities/article/34315/.

[Saw08] Sawyer, S., et al., "Social Interactions of Information Systems Development Teams: A Performance Perspective," *Information Systems Journal*, vol. 20, 2008, pp. 81–107.

[Sca00] Scacchi, W., "Understanding Software Process Redesign Using Modeling, Analysis, and Simulation," *Software Process Improvement and Practice*, Wiley, 2000, pp. 185–195, available at www.ics.uci.edu/~wscacchi/Papers/Software_Process_Redesign/SPIP-ProSim99.pdf.

[Sce02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.

[Sch01b] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001.

[Sch03] Schlickman, J., *ISO 9001: 2000 Quality Management System Design*, Artech House Publishers, 2003.

[Sch06] Schmidt, D., "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, February 2006, pp. 25–31.

[Sch09] Schumacher, R. (ed.), *Handbook of Global User Research*, Morgan-Kaufmann, 2009.

[Sch11] Schilit, B. "Mobile Computing: Looking to the Future," *IEEE Computer*, vol. 44, no. 5, May 2011. pp. 28–29.

[Sch98a] Schneider, G., and J. Winters, *Applying Use Cases*, Addison-Wesley, 1998.

[Sch98c] Schulmeyer, G., and J. McManus (eds.), *Handbook of Software Quality Assurance*, 3rd ed., Prentice Hall, 1998.

[Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics," *IEEE Trans. SE*, vol. 25, no. 6, November–December 1999, pp. 768–781, available at www.dacs.dtic.mil/topics/reliability/IEEETrans.pdf.

[SDS08] Spice Document Suite, "The SPICE and ISO Document Suite," *ISO-Spice*, 2008, available at www.isospice.com/articles/9/1/SPICE-Project/Page1.html.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates," 2003, available at http://seeri.etsu.edu/default.htm.

[SEI00] *SCAMPI, V1.0 Standard CMMI ®Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, available at www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.

[SEI02] "Maintainability Index Technique for Measuring Program Maintainability," SEI, 2002, available at www.sei.cmu.edu/str/descriptions/mitmpm_body.html.

[SEI08] "The Ideal Model," Software Engineering Institute, 2008, available at www.sei.cmu.edu/ideal/.

[SEI13] "Software Product Lines—Overview," Software Engineering Institute, 2013, available at www.sei.cmu.edu/productlines/.

[Sha05] Shalloway, A., and J. Trott, *Design Patterns Explained*, 2nd ed., Addison-Wesley, 2005.

[Sha09] Shaw, M., "Continuing Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 26, no. 8, November–December 2009, pp. 64–67.

[Sha95a] Shaw, M., and D. Garlan, "Formulations and Formalisms in Software Architecture," *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[Sha95b] Shaw, M., et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering*, vol. SE-21, no. 4, April 1995, pp. 314–335.

[Sha96] Shaw, M., and D. Garlan, *Software Architecture*, Prentice Hall, 1996.

[She10] Sheldon, F., and Vishik, C. "Moving Toward Trustworthy Systems: R&D Essentials," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 31–40.

[Shn04] Shneiderman, B., and C. Plaisant, *Designing the User Interface*, 4th ed., Addison-Wesley, 2004.

[Shn09] Shneiderman, B., et al., *Designing the User Interface*, 5th ed., Addison-Wesley, 2009.

[Shn80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.

[Sho83] Shooman, M., *Software Engineering*, McGraw-Hill, 1983.

[Shu12] Shull, F., "Designing a World at Your Fingertips: A Look at Mobile User Interfaces," *IEEE Software*, vol. 29, no. 4, July–August, 2012, pp. 4–7.

[Sin08] Sinn, R. H., *Software Security Technologies: A Programmatic Approach*, Addison-Wesley, 2008.

[Sne03] Snee, R., and R. Hoerl, *Leading Six Sigma*, Prentice Hall, 2003.

[Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems," *IEEE Software*, January 1995, pp. 24–25.

[Soa10] Soares, G., et al., "Making Program Refactoring Safer," *IEEE Software*, vol. 37, no. 4, July–August 2010, pp. 52–57.

[Soa11] Soasta, "Five Strategies for Performance Testing Mobile Applications," a white paper, 2011, available at http://info.soasta.com/performance-testing-mobile-apps-sem2.html.

[Sob10] Sobel, A., and G. McGraw, "Interview: Software Security in the Real World," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 47–53.

[Sol99] van Solingen, R., and E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.

[Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial," *IEEE Software*, vol. 22, no. 1, January–February 2005, pp. 16–23.

[Som97] Somerville, I., and P. Sawyer, *Requirements Engineering*, Wiley, 1997.

[Sou08] de Sousa, C., and D. Redmiles, "An Empirical Study of Software Developer's Management of Dependencies and Changes," *ICSE Proceedings*, May 2008, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.427&rep=rep1&type=pdf.

[Spa11] Spagnolli, B., et al., "Eco-Feedback on the Go: Motivating Energy Awareness," *IEEE Computer*, vol. 44, no. 5, May 2011, pp. 38–45.

[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction," Version 1.0, ISO/IEC JTC1, 1999.

[Spl01] Splaine, S., and S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001.

[Spo02] Spolsky, J., "The Law of Leaky Abstractions," November 2002, available at www.joelonsoftware.com/articles/LeakyAbstractions.html.

[Spr04] Spriestersbach, A., and T. Springer, "Quality Attributes in Mobile Web Application Development," 2004, available at http://atlas.tk.informatik.tu-darmstadt.de/Publications/2004/profes.pdf.

[Sri01] Sridhar, M., and N. Mandyam, "Effective Use of Data Models in Building Web Applications," 2001, available at www2002.org/CDROM/alternate/698/.

[SSO08] Software-Supportability.org, 2008, available at www.software-supportability.org/.

[Sta10] Stafford, T., and R. Poston, "Online Security Threats and Computer User Intentions," *IEEE Computer*, vol. 43, no. 1, January 2010, pp. 58–64.

[Sta97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison-Wesley, 1997.

[Sta97b] Statz, J., D. Oxley, and P. O'Toole, "Identifying and Managing Risks for Software Process Improvement," *CrossTalk*, April 1997, available at www.stsc.hill.af.mil/crosstalk/1997/04/identifying.asp.

[Ste93] Stewart, T., "Reengineering: The Hot New Managing Tool," *Fortune*, August 23, 1993, pp. 41–48.

[Sto05] Stone, D., et al., *User Interface Design and Evaluation*, Morgan Kaufman, 2005.

[Str08] Stickland, J., "How Cloud Computing Works," http://computer.howstuffworks.com/cloud-computing/cloud-computing.htm, April, 2008.

[Ste10] Stephens, M., and D. Rosenberg, *Design Driven Testing*, Apress, 2010.

[Tai89] Tai, K., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58–61.

[Tai12] Taivalsaari, A., and K. Systa, "Mobile Content as a Service: A Blueprint for a Vendor-Neutral Cloud of Mobile Devices," *IEEE Software*, vol. 29, no. 4, July–August 2012, pp. 28–33.

[Tan01] Tandler, P., "Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing," *Proceedings of UbiComp 2001: Ubiquitous Computing*, 2001.

[Tay09] Taylor, R., N. Medvidovic, and E. Dashofy, *Software Architecture*, Wiley, 2009.

[Tay90] Taylor, D., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.

[The01] Thelin, T., H. Petersson, and C. Wohlin, "Sample Driven Inspections," *Proc. of Workshop on Inspection in Software Engineering (WISE'01)*, Paris, France, July 2001, pp. 81–91. Available at http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf.

[The13] "The Emergence of Cloud Computing," 2013, available at http://www.opensourcery.com/clouddev.htm.

[Tho04] Thomas, J., et al., *Java Testing Patterns*, Wiley, 2004.

[Tho92] Thomsett, R., "The Indiana Jones School of Risk Management," *American Programmer*, vol. 5, no. 7, September 1992, pp. 10–18.

[Tic05] *TickIT*, 2005, available at www.tickit.org/.

[Tid02] Tidwell, J., "IU Patterns and Techniques," May, 2002, available at http://designinginterfaces.com/.

[Til00] Tillman, H., "Evaluating Quality on the Net," Babson College, May 30, 2000, available at www.hopetillman.com/findqual.html#2.

[Tog01] Tognozzi, B., "First Principles," *askTOG*, 2001, available at www.asktog.com/basics/firstPrinciples.html.

[Tra95] Tracz, W., "Third International Conference on Software Reuse—Summary," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 21–22.

[Tyr05] Tyree, J., and A. Akerman, "Architectural Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, no. 2, March–April, 2005.

[Uem99] Uemura, T., S. Kusumoto, and K. Inoue, "A Function Point Measurement Tool for UML Design Specifications," *Proc. of Sixth International Symposium on Software Metrics*, IEEE, November 1999, pp. 62–69.

[Ull97] Ullman, E., *Close to the Machine: Technophilia and its Discontents*, City Lights Books, 2002.

[Uni03] Unicode, Inc., *The Unicode Home Page*, 2003, available at www.unicode.org/.

[USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), January 20, 1987.

[Ute12] UTest, E-book: *Essential Guide to Mobile App Testing, 2012*, available at http://www.utest.com/landing-blog/essential-guide-mobile-app-testing.

[UXM10] "Four Best User Interface Pattern Libraries," *UX Movement.com*, available at http://uxmovement.com/resources/4-best-design-pattern-libraries/.

[Vac06] Vacca, J., *Practical Internet Security*, Springer, 2006.

[Van02] Van Steen, M., and A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[Van89] Van Vleck, T., "Three Questions About Each Bug You Find," *ACM Software Engineering Notes*, vol. 14, no. 5, July 1989, pp. 62–63.

[Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer," *Artima Developer*, December 8, 2003, available at www.artima.com/intv/contracts.html.

[Vin11] Vinson, L., "Mobile Application Testing: Process, Tools, and Techniques," 2011, available at http://threeminds.organic.com/2011/05/mobile-application-testing-process-tools-techniques.html.

[Voa12] Voas, J., et al., "Mobile Software App Takeover," *IEEE Software*, vol. 29, no. 4, July–August 2012, pp. 25–27.

[Wal03] Wallace, D., I. Raggett, and J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.

[Wal12] Walker, J., "Computer Programmers Learn Tough Lesson in Sharing," *The Wall Street Journal*, vol. 260, no. 48, August 27, 2012, p. 1.

[War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices," TMNNet, 2007, available at www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.

[War74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.

[Was10] Wasserman, A., "Software Engineering Issues for Mobile Application Development," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, 2010.

[Web05] Weber, S., *The Success of Open Source*, Harvard University Press, 2005.

[Web13] Web Application Security Consortium, 2013, available at http://www.webappsec.org/.

[Wee11] Weevers, I., "Seven Guidelines for Designing High Performance Mobile User Experiences," *Smashing Magazine*, July 18, 2011, available at http://uxdesign.smashingmagazine.com/2011/07/18/seven-guidelines-for-designing-high-performance-mobile-user-experiences/

[Wei86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.

[Wel01] vanWelie, M., "Interaction Design Patterns," 2001, available at www.welie.com/patterns/.

[Wel99] Wells, D., "XP—Unit Tests," 1999, available at www.extremeprogramming.org/rules/unittests.html.

[Wev11] Wever, A., and N. Maiden, "Requirements Analysis: The Next Generation," *IEEE Software*, vol. 28, no. 2, March–April 2011, pp. 22–23.

[Whi07] Whitehead, J., "Collaboration in Software Engineering: A Roadmap," in *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[Whi08] White, J., "Start Your Engines: Mobile Application Development," April 22, 2008, available at http://www.devx.com/SpecialReports/Article/37693.

[Whi12] Whittaker, J., et al., *How Google Tests Software*, Addison-Wesley, 2012.

[Whi97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.

[Wie02] Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.

[Wie03] Wiegers, K., *Software Requirements*, 2nd ed., Microsoft Press, 2003.

[Wik12] Wikipedia, "Data Modeling," 2012, available at http://en.wikipedia.org/wiki/Data_modeling.

[Wik13] "Cloud Computing," January 2013, available at http://en.wikipedia.org/wiki/Cloud_computing.

[Wil00] Williams, L., and R. Kessler, "All I Really Need to Know about Pair Programming I Learned in Kindergarten," *CACM*, vol. 43, no. 5, May 2000, available at http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF.

[Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software," *Computerworld*, March 21, 2005, available at www.computerworld.com/securitytopics/security/story/0,10801,91316,00.html.

[Wil97] Williams, R., J. Walker, and A. Dorofee, "Putting Risk Management into Practice," *IEEE Software*, May 1997, pp. 75–81.

[Wil99] Wilkens, T., "Earned Value, Clear and Simple," *Primavera Systems*, April 1, 1999, p. 2.

[Win90] Wing, J., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, September 1990, pp. 8–24.

[Wir71] Wirth, N., "Program Development by Stepwise Refinement," *CACM*, vol. 14, no. 4, 1971, pp. 221–227.

[Wir90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[WMT02] *Web Mapping Testbed Tutorial*, 2002, available at www.webmapping.org/vcgdocuments/vcgTutorial/.

[Woh94] Wohlin, C., and P. Runeson, "Certification of Software Components," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 494–499.

[Wor04] World Bank, *Digital Technology Risk Checklist*, 2004, available at www.moonv6.org/lists/att-0223/WWBANK_Technology_Risk_Checklist_Ver_6point1.pdf.

[Wri11] Wright, A., "Lessons Learned: Architects Are Facilitators, Too!" *IEEE Software*, vol. 28, no. 2, January–February 2011, pp. 70–72.

[W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, available at www.w3.org/TR/2003/WD-WCAG20-20030624/.

[Yac03] Yacoub, S., et al., *Pattern-Oriented Analysis and Design*, Addison-Wesley, 2003.

[Yah13] Yahoo Developer Network, Yahoo! Design Pattern Library, 2013, available at http://developer.yahoo.com/ypatterns/.

[Yau11] Yau, S., and H. An, "Software Engineering Meets Services and Cloud Computing," *IEEE Computer*, vol. 44, no. 10, October 2011, pp. 47–53.

[You01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.

[You75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, 1975.

[You95] Yourdon, E., "When Good Enough Is Best," *IEEE Software*, vol. 12, no. 3, May 1995, pp. 79–81.

[Yau02] Yaun, M. "Best Tools for Mobile Application Development," *Java World*, available at www.javaworld.com/javaworld/jw-10-2002/jw-1018-wireless.html.

[Zah90] Zahniser, R., "Building Software in Groups," *American Programmer*, vol. 3, no. 7–8, July–August 1990.

[Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, March 1994, pp. 35–38.

[Zha05] Zhang, W., and S. Jarzabek, "Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," *Proceedings of 9th Software Product Line Conference*, September 2005, pp. 57–69.

[Zim11] Zimmermann, O., "Architectural Decisions as Reusable Design Assets," *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 64–69.

[Zul92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers," *American Programmer*, February 1992, pp. 28–41.

[Zus90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.

[Zus97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

Index

Abstraction, 108, 232
Access control, 637
Accessibility, 336
Action, 16
Activity diagram, 155, 180, 303, 869
Actors, 149
Aggregate objects, 633
Aggregation, 873
Agile alliance, 66
Agile development (see also Agile process), 71
  politics of, 71
Agile manifesto, 66
Agile modeling principles, 81
Agile process, 70
Agile teams, 691
Agility,
  cost of change, 69
  definition of, 68
  principles, 70
Alpha test, 485
Analysis,
  defect, 450
  display content, 331
  rule of thumb, 169
  tasks, 326
  threat, 585
  user, 325
  user interface, 322, 325
  work environment, 331
Analysis classes,
  characteristics of, 187
  identifying, 185
  state diagrams, 204
Analysis model,
  building, 154
  elements of, 154, 172
Analysis packages, 199
Annual loss expectancy (ALE), 594
Application domains, 6
Application software, 7
Appraisal costs, 422
Archetype, 269
Architectural agility, 280
Architectural components, 270
Architectural context diagram (ACD), 267
Architectural decisions, 256, 266
Architectural description language (ADL), 233, 276
Architectural descriptions, 255
Architectural design metrics, 663
Architectural design tools, 273
Architecture, 118, 253
  alternatives, 274
  call and return, 260
  conformance checking, 279
  data centered, 258
  data flow, 259
  definition of, 253
  description template, 257
  genres, 257
  importance of, 254
  mismatch, 311
  patterns, 278
  service-oriented, 266
  structural properties, 233
  styles, 258
  tradeoffs, 275
  WebApp design, 381
Arguments for metrics, 720
Artificial intelligence software, 7
Aspect-oriented software development, 54
Aspects, 55, 237
Assessment, risk, 777
Asset, 585
Asset value, 591
Associations, 198, 872
Assurance, security, 592
Attack, 585
Attack pattern, 589
Attack surface, 596
Attributes, 188, 893
  metrics, 654, 657
  quality, 455
Audit trails, 632
Audits, software quality, 450
Automated debugging, 491
Automated estimation techniques, 743
Automated testing, 571
Availability measures, 460
Avoiding management problems, 697

Backlog, 79
Backtracking, debugging, 491
Backward impact management, 639
Baselines, 626, 720
Basis path testing, 500
Behavioral models, 203
Behavioral testing, 509
Beta test, 485
Big-bang integration testing, 476
Black box, 604
  specification, 605
  testing, 509
Bootstrap, SPI framework, 833
Bottom-up integration, 477
Boundary classes, 892
Boundary value analysis, 512
Box structure specification, 604
Brute force, debugging, 491
Bugs (See also, Faults, Defects), 432
Building blocks, 848
Business classes, 897
Business process reengineering, 799
  tools, 801
Business risk, 779

Casual meeting, 439
Cause elimination, 491
Certification, 612
  model, 612
  teams, 611
  testing, 489
  testing for MobileApps, 570
Change control, 626
  process, 636
Change management, 451, 631
Changes, types of, 8
Checklist,
  MobileApp, 570
  review, 439
  risk item, 781
Chief programmer team, 92
CK metrics suite, 667
Class-oriented metrics, 667
Class-Responsibility-Collaborator (CRC), 192
  cards, 74
Classes (also see, object), 892
  aggregate, 196
  analysis, 185
  attributes, 188
  boundary, 892
  business, 897
  characteristics, 897
  cohesion, 898
  controller, 894
  coupling, 898
  design, 239

Classes (also see, object) (*Cont.*)
diagrams of, 156, 191, 870
operations (see also, Methods), 189
persistent, 897
process, 897
testing, 481, 528
user interface, 897
Classic life cycle, 42
Cleanroom
design, 607
process model, 603
testing (see also, Statistical use testing), 610
Clear box, 605
specification, 607
Closed paradigm team, 689
Cloud computing, 10, 97, 405
Cluster testing, 482, 529
CMM, 38
CMMI, 38, 828
COCOMO II Model, 744
Code quality, 454
Code restructuring, 809
Code reviews, 433
Coding principles, 122
Cohesion, 296, 898
Collaboration, 140, 195
diagram, 880
tools, 98
Collaborative development, 852
Communication activity, 17
principles of, 110
diagram, 880
team, 692
work tasks, 696
Compatibility tests, 554
Completion, 484
Complexity,
management of, 845
elements, 626
Component model, 612
Component-based development, 53, 308
Component-based software engineering (CBSE), 308
Component-based standards, 291
Component-level metrics, 671
Component-level testing, 555
Components,
adaptation, 310
class-based, 291
classification, 312
composition, 310
definition of, 286
design guidelines, 295
design of, 290, 299
elaboration, 287
object-oriented view, 286
process-related view, 291
retrieval, 312
traditional view, 288
WebApp design, 387
Composition, 873
Concurrent models, 49
Condition testing, 507
Configuration audit, 639
Configuration management, 626
Configuration models, 220
Configuration objects, 642
Configuration review, 484
Configuration testing, 558
Connectivity testing, 483
Consistency, 526
Construction activity, 17
principles, 121
Content
model, 216
objects, 379
management, 643
repository, 630
testing, 545
testing tools, 547, 645
Context, 694
Context aware apps, 399
Contingency planning, 789
Continuous process improvement, 449
Control structure testing, 507
Controller class, 894
Conventional software test strategies, 475
Coordination and communication, 692
Coordination, team, 692
Correctness checks,
OOA model, 525
security, 591
Correctness proof, 609
Correctness verification (see also, Design verification), 608
Cost estimation tools, 748
Cost-effectiveness, 436
Coupling, 298
class, 898
CRC model review, 527
Critical practices, 699
Cross-cutting concerns, 54, 237, 589
Customer voice table, 146
Customers, 111
Myths about, 24
Cyclomatic complexity, 503

Data design, 244
Data flow modeling, 511
Data flow testing, 507
Data restructuring, 810
Data tree, 217
Database testing, 547
Debugging, 488
automated, 492
psychological considerations, 491
strategies, 491
tactics, 491
Decision tree, 749
Decomposition,
problem, 694
Decomposition techniques, estimation, 734
Defect amplification, 433
Defect analysis, 450
Defect removal efficiency (DRE), 718
Defects (also see bug, faults), 432
Dependability, 591
Dependencies, 198, 872
inversion, 241
management, 628
tracking, 631
Deployment
activity, 17, 125
diagram, 874
testing, 487
Deployment-level design, 248
Depth-first integration, 480
Design
aesthetic, 377
architectural, 267
architectural elements, 244
classes, 239
cleanroom, 607
component level, 247, 290, 299
concepts, 231
content, 379
data, 244
deployment level, 248
evolution of, 230
formal, 603
granularity, 365
interfaces, 245, 317
metrics, 663
model, 226
navigation, 385
object-oriented concepts, 238
pattern-based, 347, 354
postmodern, 854
process, 228
quality guidelines, 228, 454
refactoring, 74
refinement, 237, 608
reuse, 312
scenario-based design, 532
task set, 231
technical reviews, 229
test case, 242, 515
traditional components, 307
verification, 608
WebApps, 371
Desk check, 439
Device compatibility testing, 483

Diagram,
activity, 869
class, 870
collaboration, 880
communication, 880
deployment, 874
sequence, 876
state, 884
use-case, 875
Direct measures, 708
Document
restructuring, 804
testing, 517
Domain analysis, 170
Domain engineering, 308
Drivers, 469
Dynamic system development method (DSSD), 79

Effort estimation tools, 748
Elaboration, 135, 237
component, 287
task, 327
Elicitation, 134
agile, 148
work products, 147
Embedded Software, 7
Empirical models,
estimation, 743
Encapsulation, 892
End users, 111
Engineering,
forward, 811
reverse, 805
security, 588
Engineering/scientific software, 7
Environmental resources, 732
Equivalence partitioning, 511
Error correction, 492
Error density, 435
Errors, 432
cost of, 423
WebApp environment, 542
Estimation
agile development, 746
concepts, 750
decomposition techniques, 734
empirical models, 743
FP-based, 738
object-oriented projects, 746
problem-based, 735
problem-based, 735
process-based, 739
reconciliation, 742
risk (see also, Risk projection), 782
software, 727
use case, 740
WebApp, 747
Evaluations, post-mortem, 445
Exceptions, 178
Exhaustive testing, 500
Exposure, risk, 786
Extreme programming (XP), 72
activities, 72
industrial, 75
testing, 75
team, 94

Failure costs, 422
Failure curve,
hardware, 5
software, 6
Fault (see also, Bug, Defects), 432
Fault-based testing, 531
Feasibility analysis,
MobileApp, 571
Finite state modeling, 511
Fire-fighting, 778
Flow graph, 500
Formal design, 603
Formal methods, 53, 602
tools, 614
Formal specification language, 900
Formal technical review (FTR), 432, 441
Forward engineering, 811
Forward impact management, 638
Framework, 17, 291, 351
activities, 16, 17, 32
SPI, 819
Function point (FP), 659, 710
estimation, 738
Functional decomposition, 694
Functional independence, 236
Functional model, 218
Functional testing, 509

Gap analysis, 823
Generalization, 871
Genres, 257
Gesture testing, 575
Goal/question/metric (GQM), 656
Graph matrices, 506
Graph-based testing, 509
Graphic design, 378

Hazards, 463, 790
Help facilities testing, 516
Historical data, 659
Horizontal refinement, 613
Human elements, 626
Human resources, 731
Hype cycle, 843

Identification, 633
risk, 780
Impact management, 638
risk, 785
Impact of risk, 785
Inception, 133
increments, 44
Independent program paths, 502
Independent test group (ITG), 475
Indicator, 655
Informal reviews, 439
Information, 3
objectives, 694
representing, 862
Information hiding, 235
Information objectives, 694
Inheritance, 894
Inspections, 432, 437
Integration testing, 475
object-oriented, 529
work products, 480
Integration,
bottom-up, 481
top-down, 473
Interaction frames, 878
Interclass test-case design, 534
Interface analysis, 325
Interface design, 245, 317
evaluation of, 342
golden rules, 318
issues, 335
MobileApps, 341, 399
models, 323
patterns, 362
process, 323
steps, 332
WebApps, 337, 376
Internationalization, 336, 578
Internet of things, 588
Invariant, 616
Inventory analysis, 803
ISO 9001:2000, 38
ISO 9001:2008, 462

Language
formal specification, 900
object constraint, 887
semantics, 901
syntax, 901
z-specification, 904
Layout, 378
Legacy software, 7
Liability, 425
Living models, 120
Load testing, 562
LOC-based metrics, 712
Loop testing, 507
Lorenz and Kidd OO metrics, 671

Maintainability, 797
Maintenance, 108
software, 796
Make/Buy decision, 748

Management,
myths, 23
project, 684
risk, 451, 777
security, 451
spectrum, 685
Manual test integration tools, 614
Maturity level, 831
Maturity models, 821
Measurement, 654, 708
principles, 656
Measures, 654
availability, 461
direct, 708
direct, 708
reliability, 460
Meetings,
casual, 439
review, 441
Melding problem and process, 695
Messages, 895
Methods (see also, Operations), 16, 893
Metrics, 654
architectural design, 663
arguments, 720
attributes, 654, 657
baseline, 720
business goals, 724
collection, 721
small organizations, 721
program establishment, 722
class-oriented, 667
component-level, 671
design, 663
function-based, 659, 710
LOC-based, 712
MobileApp design, 673
object-oriented, 666, 713
private, 706
process, 704
productivity, 712
project, 707
public, 706
quality, 456
requirements model, 659
reviews, 534
size-oriented, 709
software quality, 716
source code, 675
specification quality, 663
testing, 676
use case, 714
user interface design, 672
WebApp, 714
WebApp design, 673
Middleware, 405
MobileApps (mobile applications), 9
architectural design of, 274
best design practices, 401
checklist, 570
component-level design, 306
context aware, 399
design, 391
design challenges, 392
design mistakes, 401
design metrics, 673
development activities, 395
interface design, 398
patterns, 366
quality checklist, 397
real-time testing, 578
requirements modeling, 214
SCM, 640
software engineering, 407
stress testing, 573
test matrix, 572
testing, 483
testing guidelines, 568
testing strategies, 569
testing tools, 579
tools, 404
usability testing, 575
Mobility environments, 403
Model-based testing, 514
Model-driven software development, 853
Model-View-Controller (MVC), 384
Model,
certification, 612
COCOMO II, 744
component, 612
sampling, 612
security, 590
Modeling, 17
agile, 80
CRC, 192
data flow, 511
finite state, 511
principles, 114
scenario-based, 173
security, 590
threat, 594
timing, 511
transaction, 511
Models
behavioral, 203
design, 243
object-oriented, 525
requirements, 167
Modularity, 234
Module (see also, Component), 288
MOI leadership model, 688
Mongolian horde concept, 24
MOOD metrics suite, 670
Multiple class testing, 534
Multiplicity, 872
Myths, 23
customer, 24
management, 23
practitioner, 25

Navigation,
semantic unit, 387
semantics, 385
syntax, 387
Navigation modeling, 220
Navigation testing, 556
Negligence, 425
Negotiation, 135, 159

Object (see also, Class), 892
Object constraint language (OCL), 887, 901
example of, 903
notation, 902
Object-oriented
analysis, 172, 184
design metrics, 666
integration testing, 529
metrics, 713
model consistency, 526
models, 525
OOA model correctness, 525
OOD model correctness, 525
project estimation, 746
software, 481
test case design, 530
testing strategies, 528
unit testing, 528
validation testing, 529
OO metrics, Lorenzz and Kidd, 671
Open paradigm team, 689
Open source, 848
Open world software, 846
Open-closed principle, 292
Operations (see also Methods), 189, 893
Organizational paradigms, 92
Orthogonal array testing, 513
Outsourcing, 750

Pair programming, 75, 440
Partition testing, 533
Partitioning, 694
Pattern organizing table, 358
Patterns, 109
analysis, 157, 207
architectural, 263, 278, 359
component-level, 360
context, 348
describing, 352
design, 348
example of, 209
interface design, 362
kinds of, 349
languages, 353
MobileApps, 366
process, 35
repository, 353, 360

requirements modeling, 207
template for, 352
testing, 519
WebApps, 364
Peer reviews, 432
People, 687
People Capability Maturity Model, 685, 832
People software process (PSP), 59, 833
Performance testing, 487, 580
Persistent classes, 897
Phishing, 587
Plan, RMMM, 790
Plan, SQA,
Planning activity, 17
principles, 112
web testing, 543
Plug points, 351
Polymorphism, 896
Post-mortem evaluations (PME), 445
Postconditions, 616
Practice (software engineering),
core principles, 108
essence of, 19
Pre-condition, 178, 616
Predictable risk, 779
Prevention costs, 422
Priority points, 140
Privacy, 586
cloud computing, 587
social media, 587
Private metrics, 706
Proactive risk strategies, 778
Problem decomposition, 694
Problem elaboration, 694
Problem solving, 19
Problem-based estimation, 735
Process, 16, 686
adaptation, 18
agile, 69
assessment, 37
classes, 897
components of, 16
debugging, 490
decomposition, 694
elements, 626
extreme programming, 72
flow, 31, 33
framework, 17, 32
generic, 31
iterative, 31
immaturity, 822
pattern template, 35
patterns, 35, 37
principles, 106
relationship to product, 62
SCM, 632
Process improvement, 37, 818
approaches, 819
continuous, 449
education, 825
evaluation, 827
gap analysis, 823
installation, 826
justification, 825
other frameworks, 832
ROI, 834
steps, 823
trends, 835
Process maturity, 821
Process metrics, 704
Process modeling, tools, 62
Process models, 40
cleanroom, 603
concurrent, 49, 51
evolutionary, 45
incremental, 43
prescriptive, 41
risk driven, 48
software reengineering, 803
specialized, 52
Process technology, 61
Process-based estimation, 339
Producer, 441
Product, 686, 693
Product metrics, 653
tools, 678
Product-line software, 7, 11
Product-specific risk, 780
Productivity measures, 712
Productivity metrics, 712
Project, 686, 693
Project complexity, 728
Project database, 627
Project management, 684
tools, 699
Project metrics, 707
Project planning, 729
process, 729
task set, 730
Project risk, 779
Project size, 729
Project velocity, 73
Projects, getting started, 26
Projection, risk (see also, Risk estimation), 782
Prototyping, 45
problems with, 46
Public metrics, 706
Public metrics, 706

Quality,
attributes, 455
"good enough", 421
code, 455
concepts, 412
cost of, 422
definition of, 413
design, 454
Garvin's dimensions, 415
ISO 9126, 418
management actions, 426
McCall's factors, 416
methods for achieving, 427
metrics, 455
MobileApps, 397
quantitative view, 420
requirements, 454
requirements tree, 373
security, 425
WebApp design, 372
Quality assurance, 428
statistical, 456
Quality control, 427
Quality function deployment (QFD), 146
Quality management, 449
resources, 452

Random paradigm team, 689
Random testing, 532
Rapid cycle testing, 473
Reactive risk strategies, 778
Real-time testing,
MobileApps, 578
System, 517
Reconciliation,
estimation, 742
LOC and FP metrics, 711
Record keeping, 442
Recorder, 442
Recovery testing, 486
Reengineering
business process, 799
economics, 813
software, 802
Refactoring, 74, 238, 301
Refinement, (see also, Elaboration), 237
Regression testing, 478
Reliability,
measures, 459
software, 459
Repository, 630
content, 630
design patterns, 360
hypermedia, 365
SCM, 630
Requirements,
emergent, 846
negotiating, 159
nonfunctional, 141
security, 585
understanding, 131
validation, 161
Requirements elicitation, security, 589
Requirements engineering, 132, 852
agile, 158
common mistakes, 162
first questions, 140
goal-oriented, 134
tools, 138

Requirements gathering, collaborative, 143
Requirements modeling,
approaches to, 171
objectives, 168
principles, 116
WebApps, 213
MobileApps, 213
Requirements models (see also, Analysis models), 114
metrics, 659
types of, 167
Requirements quality, 454
Requirements specification, template for, 136
Requirements tasks,
elaboration, 135
elicitation, 134, 142
inception, 133
negotiation, 135
specification, 135
validation, 136
Requirements tracing, 631
Resources, 731
environmental, 732
human, 731
quality management, 452
reusable software, 732
Responsibilities, 193
Restructuring, 809
code, 809
data, 810
document, 804
Reusable software resources, 732
Reuse, 312
Reverse engineering, 805
data, 807
processing, 807
tools, 809
user interfaces, 808
Review
checklist, 439
guidelines, 442
issues list, 442
leader, 441
meeting, 441
metrics, 435
reporting, 442
Review information sheet (RIS), 790
Reviews,
architectural, 277
code, 433
configuration, 488
informal, 439
peer, 432
sample-driven, 444
software quality, 450
technical, 441
Risk
assessment, 777
categories, 779
components, 782
drivers, 782
estimation (see also risk projection), 782
exposure, 786
identification, 780
impact, 785
information sheet, 790
item checklist, 781
patterns, 781
projection (see also risk estimation), 782
refinement, 787
Risk management, 777
principles, 780
tools, 791
SPI, 827
Risk Mitigation, Monitoring, and Management (RMMM), 788, 790
Risk strategies,
proactive, 778
reactive, 778
Risk table, 783
Risks, 424
business, 779
known, 779
predictable, 779
product-specific, 780
project, 779
technical, 779
unpredictable, 779

SafeHome, 143, 150
activity diagram, 220
analysis patterns, 209
applying patterns, 362
architectural assessment, 276
architectural decisions, 265
architecture context diagram, 269
behavioral modeling, 157
choosing an architecture, 262
class models, 190
cohesion in action, 297
communication mistakes, 111
conclusion, 862
considering agile process, 76
coupling in action, 298
CRC models, 197
data tree, 217
design classes, 241
design concepts, 239
design patterns, 356
design vs. coding, 227
domain analysis, 171
getting started, 26
grammatical parse, 186
graphic design, 377
instantiations of, 272
interface design review, 340
interface golden rules, 320
MobileApp requirements, 396
negotiation, 160
open closed principle, 293
preliminary user scenario, 174
quality issues, 424
requirements gathering, 145
screen layout, 334
selecting a model, 47, 50
sequence diagram, 206
state diagram, 205
swimlane diagram, 181
team structure, 93
use case diagram, 153
use case for UI design, 327
use case template, 178
user scenarios, 147
Safety, 451, 790
software, 460
Sample-driven reviews, 444
Sampling model, 612
Scenario-based testing, 532
Scope, 113, 134
software, 694
Scrum, 78
meetings, 79
Security and privacy, 586
Security assurance, 592
Security assurance case (see also, Trustworthiness), 592
Security engineering, 588
tools, 598
user-centered, 589
Security
case, 591
concerns, cross-cutting, 589
management, 451
correctness checks, 591
model, 590
objectives, 590
requirements elicitation, 585, 589
quality aspects, 425
Self-adaptive systems, 158
Self-organizing teams, 692
Semantics, language, 901
Separation of concerns, 108, 234
Sequence diagrams, 205, 876
Service-oriented methods, 148
Services, 893
Six sigma, 458
Size-oriented metrics, 709
Smoke testing, 479
Social media, 95
Social media and privacy, 587
Software architecture, see Architecture

Software component, see Component
Software configuration audit, 639
Software configuration items (SCI), 628
Software configuration management (SCM), 623
  MobileApps, 640
  process, 632
  Repository, 630
  standards, 649
  tools, 640
  WebApps, 640
Software defects, cost impact, 432
Software design, see Design,
Software development, myths, 23
Software engineer,
  characteristic of, 88
  roles, 89
Software engineering,
  cloud-based, 97
  component-based, 308
  core concepts, 106
  definition of, 15
  design, 225
  ethics, 865
  grand challenge, 851
  impact of social media, 95
  layers of, 16
  long view, 864
  practice, 19
  principles, 21, 104
  psychology, 89
  technology directions, 849
  trends, 839
  video games, 1
  work practices, 126
Software equation, 744
Software estimation, 727
Software maintenance, (see also, Maintenance), 796
Software Process, 16
Software process improvement (SPI), see Process improvement
Software quality assurance (SQA), 448
  elements of, 450
Software quality, see also, Quality
  audits, 450
  metrics, 716
  reviews, 450
  standards, 450
Software reengineering, 802
  process model, 803
Software reliability, 459
Software restructuring tools, 810
Software safety, 460
Software scope, (see also, Scope), 694, 730
Software sizing, 734
Software teams, (see also, Teams), 689
Software,
  application domains, 6
  as capital, 30
  definition of, 4
  "good enough", 421
  importance of, 2, 861
  key questions, 4
  nature of, 3
  object-oriented, 487
  realities, 14
Source code metrics, 675
Specification, 135
  black-box, 605
  box structure, 604
  clear-box, 607
  quality metrics, 663
  state-box, 606
SPICE, 38, 833
Spike solution, 74
Spiral model, 47
Sprints, 79
SQA group, 450
SQA plan, 454
Stakeholder, 687
  definition of, 17
  identifying, 139
Standards,
  ISO 9001:2008, 462
  SCM, 649
  software quality, 450
State box, 604
State diagram, 884
State representations, 204
State-box specification, 606
Statechart, 304
Statistical quality assurance, 456
Statistical software process improvement (SSPI), 707
Statistical use testing, 604
Status reporting, configuration, 639
Stereotype, 871
Strategy,
  debugging, 491
  testing, 471
Stress testing, 487, 562
  MobileApp, 573
Structure chart, 289
Structured analysis, 171
Stub, 486
Subclass, 893
Superclass, 893
Supportability, 798
Surface, attack, 596
Survivability, 591
Swimlane diagram, 181, 330
Synchronization control, 637
Synchronous paradigm team, 690
Syntax, 901
System classes, 897
System of forces, 348
System software, 6
System testing, 486
System vulnerability, 591

Task, 16
Task analysis, 326
Task set, 31
  identification of, 34
  project planning, 730
  project planning, 730
Team leader, 688
Team Software Process (TSP), 60, 833
Teams, 90
  agile, 93, 691
  closed paradigm, 689
  global, 99
  jelled, 90
  open paradigm, 689
  organizational paradigms, 92
  random paradigm, 689
  self-organizing, 94, 692
  structures, 92
  synchronous paradigm, 690
  talent mix, 847
  toxicity, 91
  XP, 94
Technical review reference model, 438
Technical reviews, 441
Technical risk, 779
Technology evolution, 840
Test across borders (also see, Internationalization), 578
Test case derivation, 504
Test case design, 515
Test,
  alpha, 485
  beta, 477
  characteristics, 498
  extraction tools, 614
  generation tools, 614
  object-oriented, 530
  strategies for conventional software, 473
Test matrix, 572
Test-driven development, 854
Testability, 497
Testing
  automated, 571
  basis path, 500

Testing (*Cont.*)
behavioral, 509
black-box, 509
certification, 570
class, 482, 528
cleanroom (see also Statistical use testing), 610
cluster, 529, 472
component-level, 555
condition, 507
control structure, 507
criteria, 472
data flow, 507
database, 547
deployment, 475
documentation, 517
exhaustive, 500
fault-based, 531
functional, 509
gesture, 575
graph-based, 509
guidelines, 568
help facilities, 516
integration, 481
loop, 507
methods, 529, 676
mobile-app, 482
model-based, 514
multiple class, 534
object-oriented, 528
orthogonal array, 513
partition, 533
patterns, 519
performance, 486
principles, 123
process, 544
Testing, random, 532
real-time system, 517
recovery, 478
regression, 486
scenario-based, 532
security, 479
smoke, 487
statistical use, 604
strategies, 528, 543, 569, 470
stress, 475
system, 483
thread-based, 476
thread-based, 529
tools, 579
unit, 468
use-based, 529
validation, 468
WebApp, 482
white-box, 500
Testing-in-the-wild, 483, 573
Thread-based testing, 482, 529
Threat
analysis, 585
likelihood, 591
modeling, 594
TickIt, 834
Timing modeling, 511
Tools,
agile process,
architectural description languages, 277
architectural design, 273
architecural conformance, 280
business process reengineering, 801
CBSE, 313
collaboration, 98
content management, 645
cost estimation, 748
effort and cost estimation, 748
effort estimation, 748
formal methods, 614
manual test integration, 614
MobileApp testing, 580
process modeling, 62
product metrics, 678
project and process metrics, 716
project management, 699
requirements engineering, 138
reverse engineering, 809
risk management, 791
SCM, 640
security engineering, 598
software restructuring, 810
test extraction, 614
test generation, 614
use cases, 154
user interface development, 337
user-interface testing, 554
version control, 635
web configuration testing, 559
web content testing, 547
web navigation tools, 557
web performance testing, 563
web security testing, 560
WebApp change management, 647
WebApp Metrics, 675
Top-down integration, 476
Toxicity, team, 690
Traceability, 118, 142
matrix, 142, 628
Tracking,
dependency, 631
Transaction modeling, 511
Trends,
process, 849
tools, 855
Trigger, 178
Trust verification, 591
Trustworthiness, 591
Umbrella activities, 17
UML Notation, 869
activity diagram, 155, 180, 303
analysis modeling, 207
class diagram, 156
deployment diagram, 248
interface representation, 246
state diagram, 156
swimlane diagram, 181
use case diagram, 153
Uncertainty, 779
Unified Process, 55
agile, 82
history of, 56
phases, 56
workflow, 58
Unit testing, 473
object-oriented, 528
Usability, 322
testing, 552, 575
Usage scenarios, 146
Use-based testing, 529
Use case, 875
estimation, 740
Use case points (UCP's), 742
Use cases, 146, 326
creating, 173
developing, 149
diagram, 153, 179, 875
estimation, 740
events, 203
formal, 177
metrics, 714
question to be answered, 150
refinement of, 176
tools, 154
User interface classes, 897
User interface design, see Interface design,
User interface design metrics, 672
User interface testing, 549
User stories, 73
User-centered security engineering, 589
User experience testing, 483

V-model, 43
Validation, 136, 161, 483
Validation testing, 483
checklist, 137
object-oriented, 529
Value adjustment factors (VAF), 660
Verification, 470
correctness (see also design verification), 608
design (see also design verification), 608
trust, 591

Version control, 634
tools, 635
Versioning, 631
Vertical refinement, 613
Viewpoints, multiple, 139
Virtual keyboard input, 577
Voice input, 576
Voice recognition, 576
Volatility, 134

W^5HH principle, 698
Walkthroughs, 432
Waterfall model, 41
problems with, 42
Web
configuration testing tools, 559
navigation testing tools, 557
performance testing tools, 563
quality dimensions, 541
security testing tools, 560
test planning, 543
testing process, 544
testing strategy, 543
user-interface (UI) testing tools, 554
Web/mobile applications, 7
WebApps (Web applications), 9
architectural design of, 273
architecture, 381
change management tools, 647
characteristics, 379
component design, 387
component-level design, 305
content model, 216
design, 371
design goals, 374
design metrics, 673
environment errors, 542
Estimation, 747
metrics, 714
metrics tools, 675
SCM, 640
design pyramid, 375
functional model, 218
interface design, 376
managing change, 647
navigation modeling, 220
quality of, 372
requirements modeling, 213
testing, 482
White-box testing, 500
Work practices, 126
Work products, integration testing, 483
Work tasks, communication, 696
Workflow analysis, 328

Z specification language, 904
example of, 906
notation, 905

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank