CHAPTER

22 SOFTWARE TESTING STRATEGIES

KEY CONCEPTS

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?"

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.

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."

Quote:

"Testing is the unavoidable part of any responsible effort to develop a software system."

William Howden

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.

22.1.2 Organizing for Software Testing

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.

Quote:

"Optimism is the occupational hazard of programming; testing is the treatment."

Kent Beck

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.

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.

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.

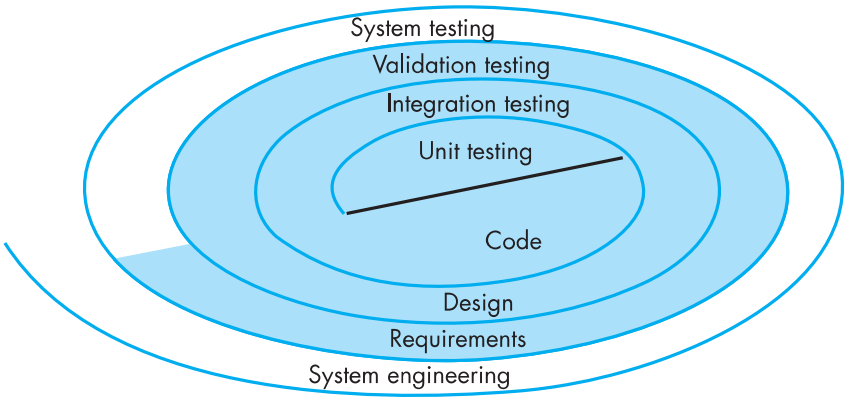
"The first mistake that people make is thinking that the testing team is responsible for assuring quality."

Brian Marick

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?

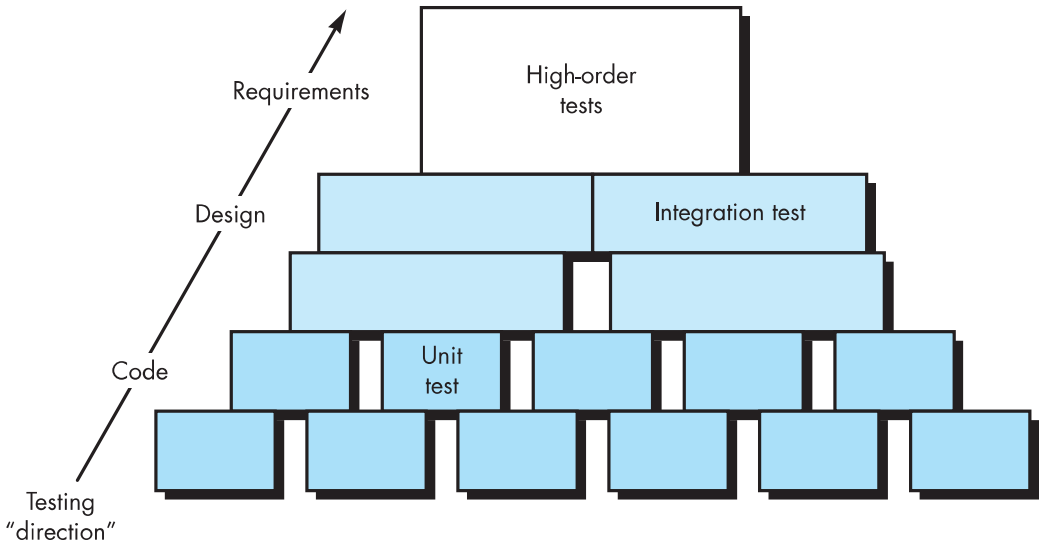
A strategy for software testing may also be viewed in the context of the spiral (Figure 22.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving 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.