TESTING STRATEGIES

Sameera Abu Ghalyoun
PPU

INDEX

Strategic Approach to Software Testing

- To perform effective testing, a software team should conduct effective formal technical reviews.
- Testing begins at the component level and work outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) by an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

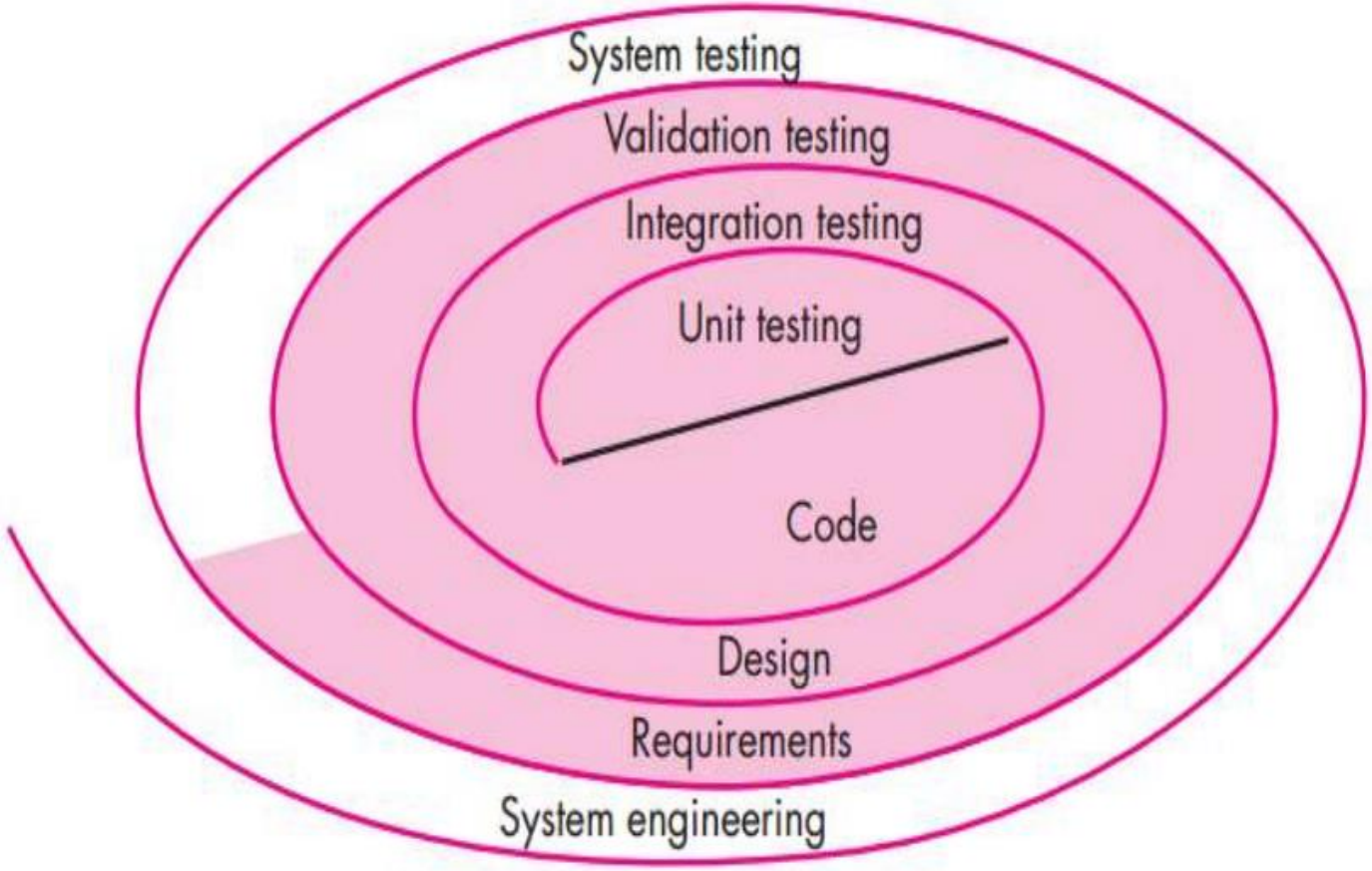
Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?) – The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?) – The set of activities that ensure that the software that has been built is traceable to customer requirements.
- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

Organizing for Software Testing

- Testing should aim at "breaking" the software.
- Common misconceptions – The developer of software should do no testing at all
- that the software should be “tossed over the wall” to strangers who will test it mercilessly.
- that testers get involved with the project only when the testing steps are about to begin.
- software architecture is complete does an independent test group become involved.
- 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.

Testing Strategy



Criteria for Completion of Testing

when is testing completed ??

- 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.
- By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: “When are we done testing?”

Strategic Issues

1

Specify product requirements in a quantifiable manner long before testing commences.

2

State testing objectives explicitly.

3

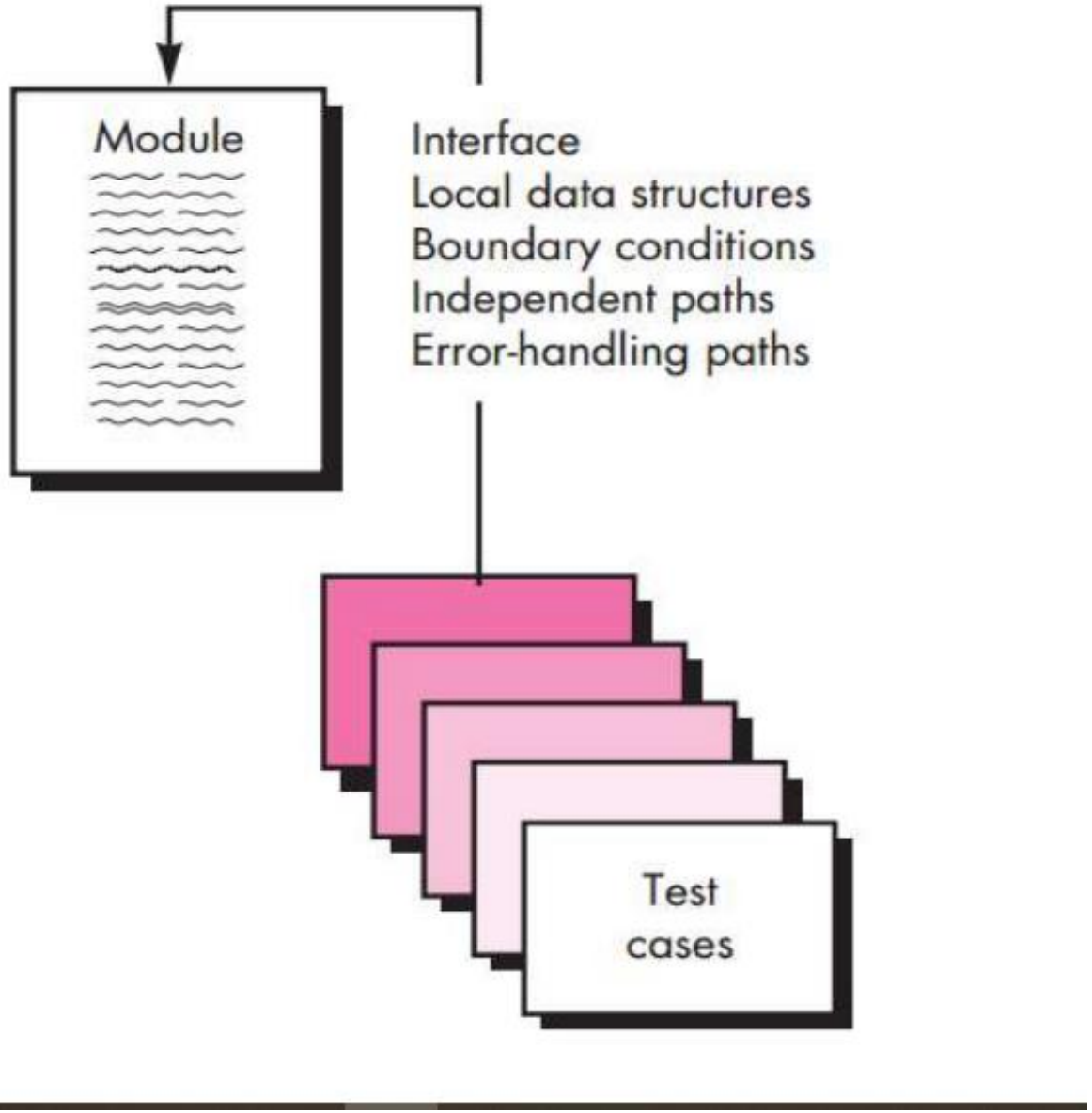
Understand the users of the software and develop a profile for each user category.

4

Develop a testing plan that emphasizes “rapid cycle testing.”

Test Strategies for Conventional Software

Unit testing	Focuses testing on the function or software module.
	Concentrates on the internal processing logic and data structures.
	Is simplified when a module is designed with high cohesion – Reduces the number of test cases – Allows errors to be more easily predicted and uncovered.
	Concentrates on critical modules and those with high cyclomatic complexity when testing resources are limited.



Unit Testing

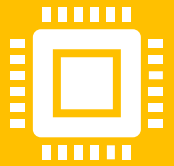
Unit testing considerations

- Module interface – Ensure that information flows properly into and out of the module.
- Local data structures – Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution.
- Boundary conditions – Ensure that the module operates properly at boundary values established to limit or restrict processing.
- Independent paths (basis paths) – Paths are exercised to ensure that all statements in a module have been executed at least once.
- Error handling paths – Ensure that the algorithms respond correctly to specific error conditions.

Unit test procedures

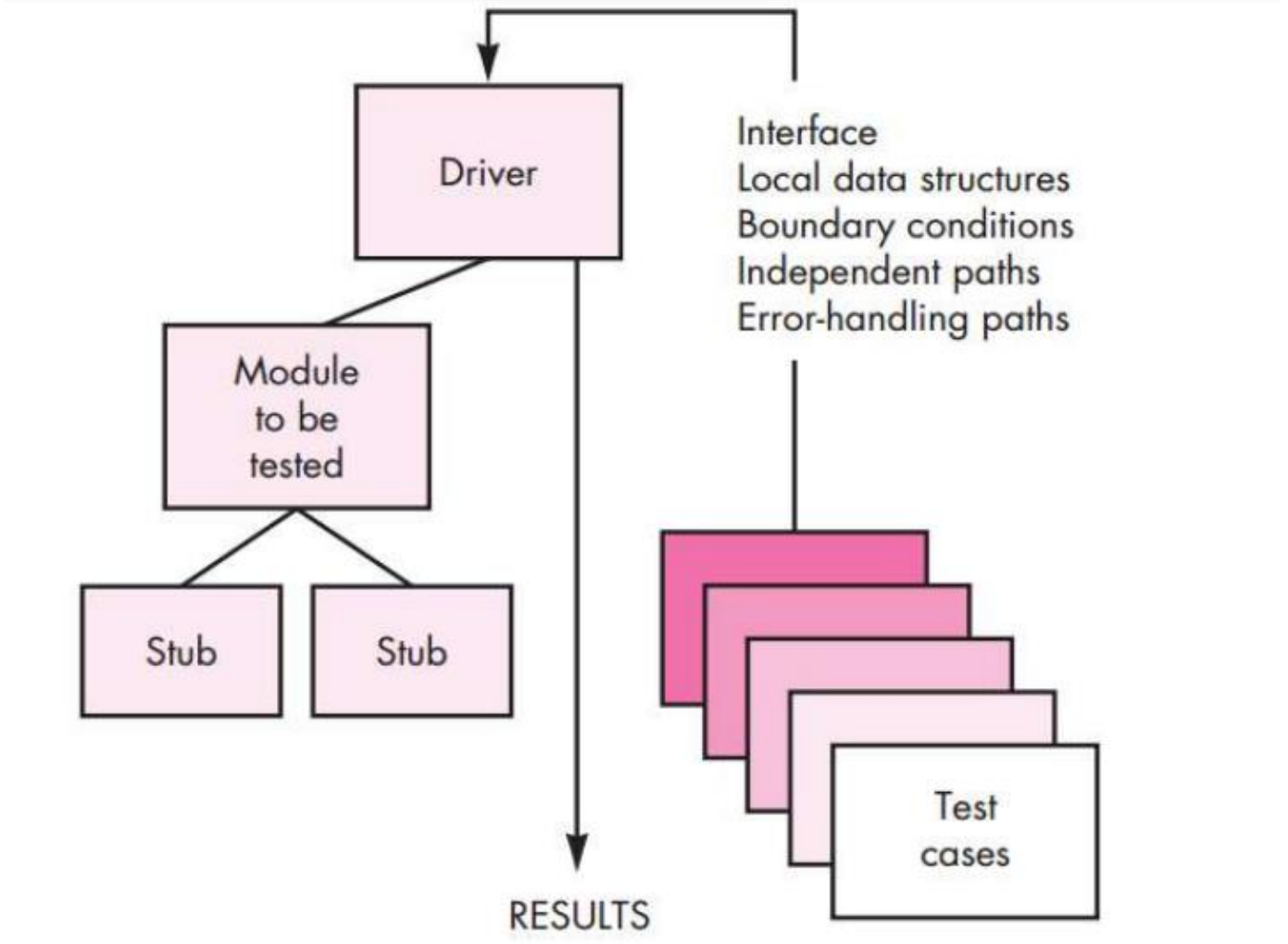
Driver – A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results.

Stubs – Serve to replace modules that are subordinate to (called by) the component to be tested – It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing.



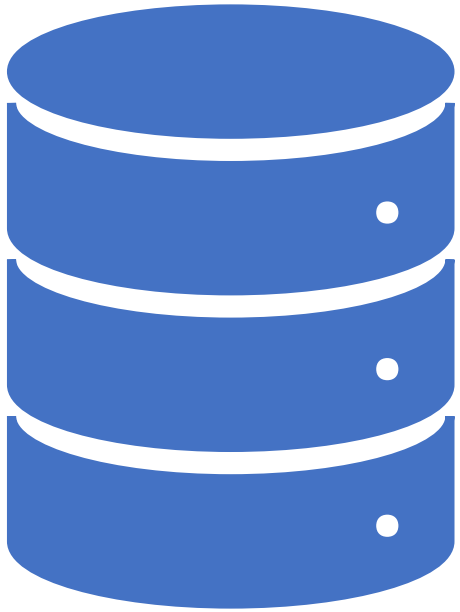
Drivers and stubs both represent testing overhead. – Both must be written but don't constitute part of the installed software product.

Unit Test Environment



Integration testing

- Defined as a systematic technique for constructing the software architecture – At the same time integration is occurring, conduct tests to uncover errors associated with interfaces.
- Objective is to take unit tested modules and build a program structure based on the prescribed design.
- Two Approaches
  - Non-incremental Integration Testing.
  - Incremental Integration Testing.



Non-incremental Integration Testing

- Uses “Big Bang” approach.
- All components are combined in advance.
- The entire program is tested as a whole Chaos results.
- Many seemingly-unrelated errors are encountered.
- Correction is difficult because isolation of causes is complicated.
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop.

Incremental Integration Testing

- The program is constructed and tested in small increments.
- Errors are easier to isolate and correct.
- Interfaces are more likely to be tested completely.
- A systematic test approach is applied.
- Different incremental integration strategies
  - Top-down integration
  - Bottom-up integration
  - Regression testing
  - Smoke testing

Top-down Integration

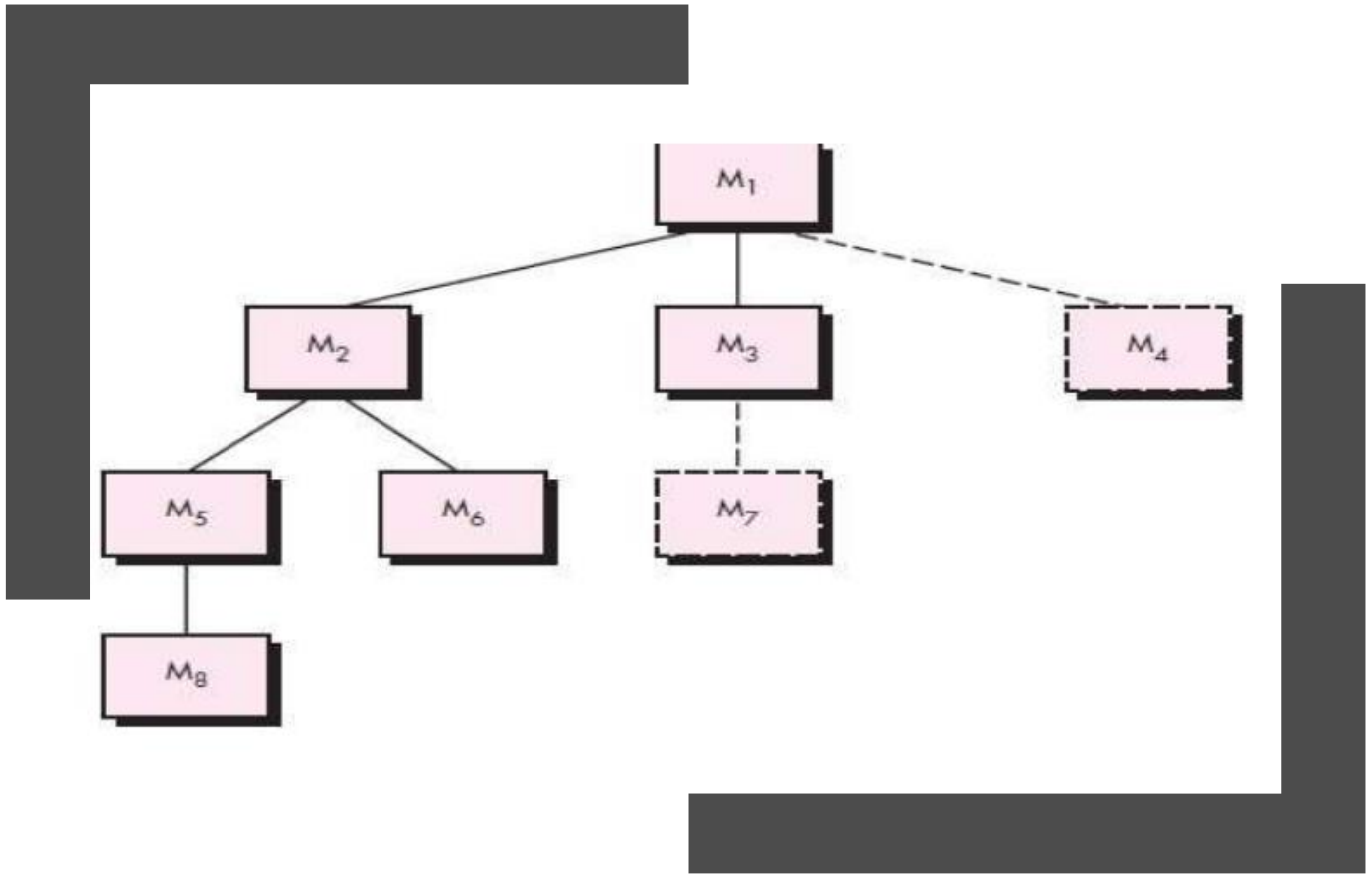
- Modules are integrated by moving downward through the control hierarchy, beginning with the main module.
- Subordinate modules are incorporated in two ways : – depth-first : All modules on a major control path are integrated – breadth-first : All modules directly subordinate at each level are integrated.
- Advantages:

– This approach verifies major control or decision points early in the test process.

- Disadvantages:

– Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded.

– Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

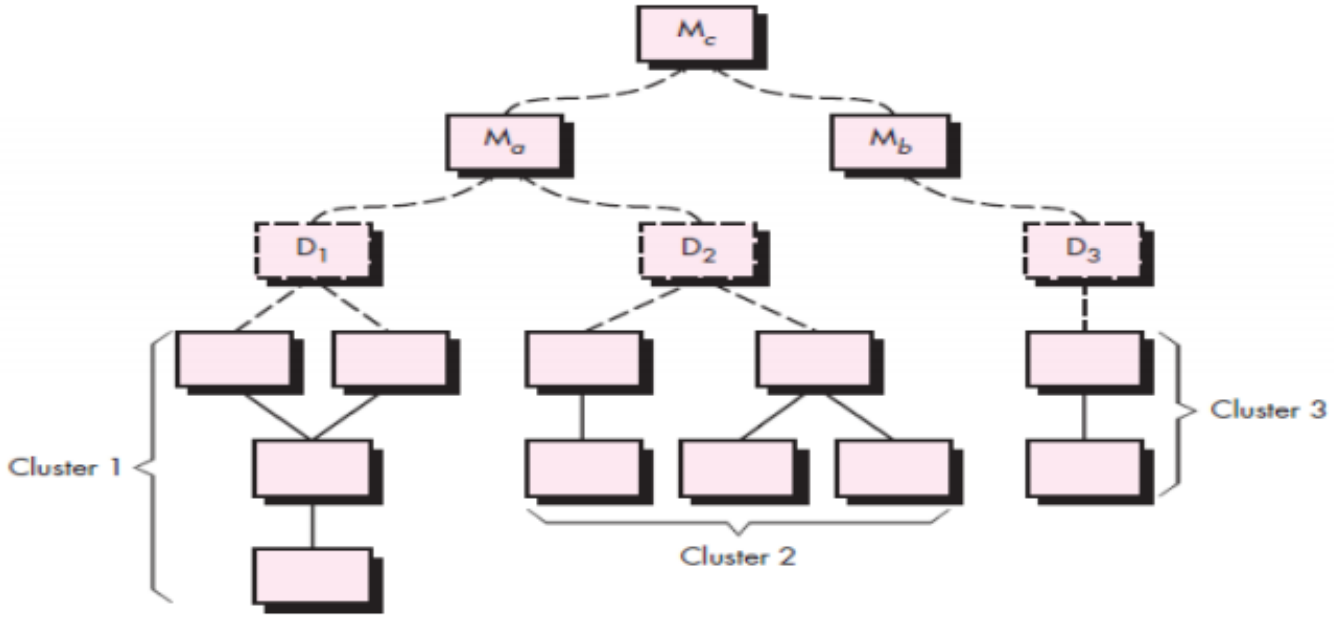


- For example, selecting the left-hand path, components M1, M2, M5 would be integrated first.
- Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated.
- Then, the central and right-hand control paths are built.

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy.
- Advantages
  - This approach verifies low-level data processing early in the testing process – Need for stubs is eliminated.
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version.
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available.

- Integration follows the pattern illustrated in Figure 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 Ma. Drivers D1and D2 are removed and the clusters are interfaced directly to Ma.
- Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth. 2



Regression Testing

- Each new addition or modification of data may cause problems with functions that previously worked flawlessly.
- Regression testing re-executes a small subset of tests that have already been conducted:

– Ensures that changes have not propagated unintended side effects.

– Helps to ensure that changes do not introduce unintended behavior or additional errors.

– May be done manually or through the use of automated capture/playback tools.

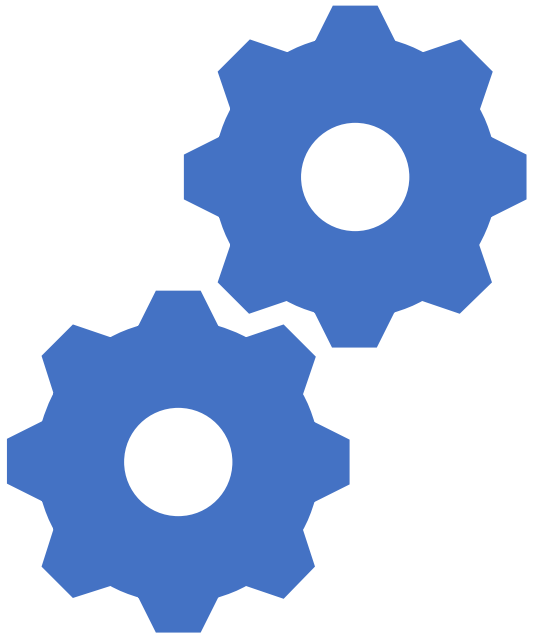
• Regression test suite contains three different classes of test cases:

– 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 actual software components that have been changed

Smoke testing



- Designed as a pacing mechanism for time-critical projects

– Allows the software team to assess its project on a frequent basis.

- Includes the following activities:

– The software components that have been translated into code and linked into a build.

– A series of breadth tests is designed to expose errors that will keep the build from properly performing its function.

- The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.

– The build is integrated with other builds and the entire product is smoke tested daily.

Test Strategies for Object-Oriented Software

Unit testing in OO context :

- **Class testing for object-oriented software is**
  - the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- **Integration testing in OO context**: Two different object-oriented integration testing **strategies are:**

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.

Test Strategies for Object-Oriented Software

Integration test in OO context:

- Second :-Regression testing is applied to ensure that no side effects occur – Use-based testing

- First tests the independent classes that use very few, if any, server classes.
- Then the next layer of classes, called dependent classes, are integrated.
- This sequence of testing layer of dependent classes continues until the entire system is constructed.

Test Strategies for Web Apps

- 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:

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

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 reliability and performance.

Validation Testing

- **Validation testing follows integration testing.**
- The distinction between conventional and object-oriented software disappears and Focuses on user-visible actions and user-recognizable output from the system.

Validation test criteria :

- Demonstrates conformity with requirements.
- Designed to ensure that All functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained.
- Documentation is correct.
- Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
- **After each validation test**
  - The function or performance characteristic conforms to specification and is accepted.
  - A deviation from specification is uncovered and a deficiency list is created

Configuration review:

• The intent of this 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.

Alpha and beta testing :

• Alpha testing conducted at the developer's site by end users

– Software is used in a natural setting with developers watching intently.

– Testing is conducted in a controlled environment.

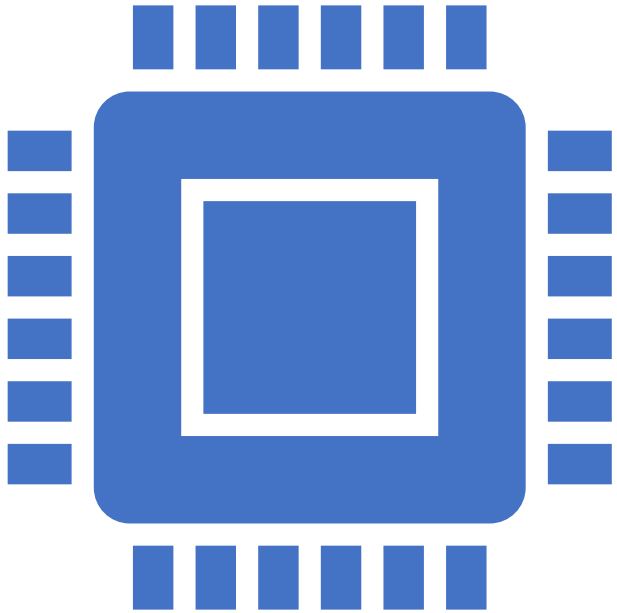
• Beta testing conducted at end-user sites:

– Developer is generally not present.

– It serves as a live application of the software in an environment that cannot be controlled by the developer.

– The end-user records all problems that are encountered and reports these to the developers at regular intervals.

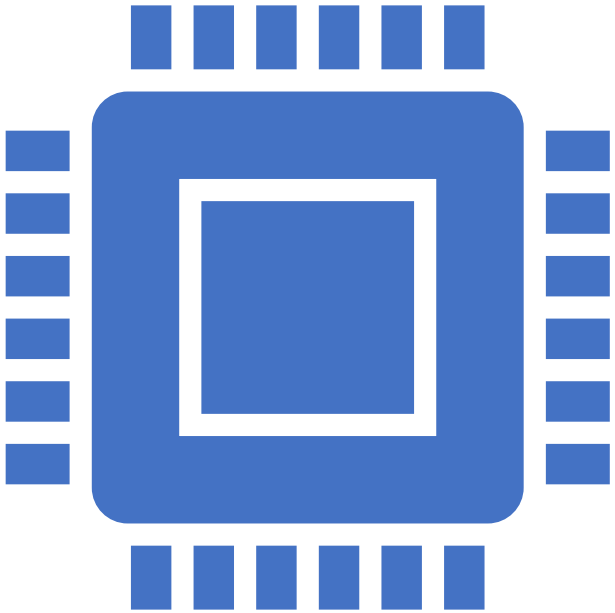
• After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base.



System Testing

- Software may be part of a larger system. This often leads to “finger pointing” by other system dev teams.
- Finger pointing defense:
  1. Design error-handling paths that test external information.
  2. Conduct a series of tests that simulate bad data.
  3. Record the results of tests to use as evidence.

System Testing



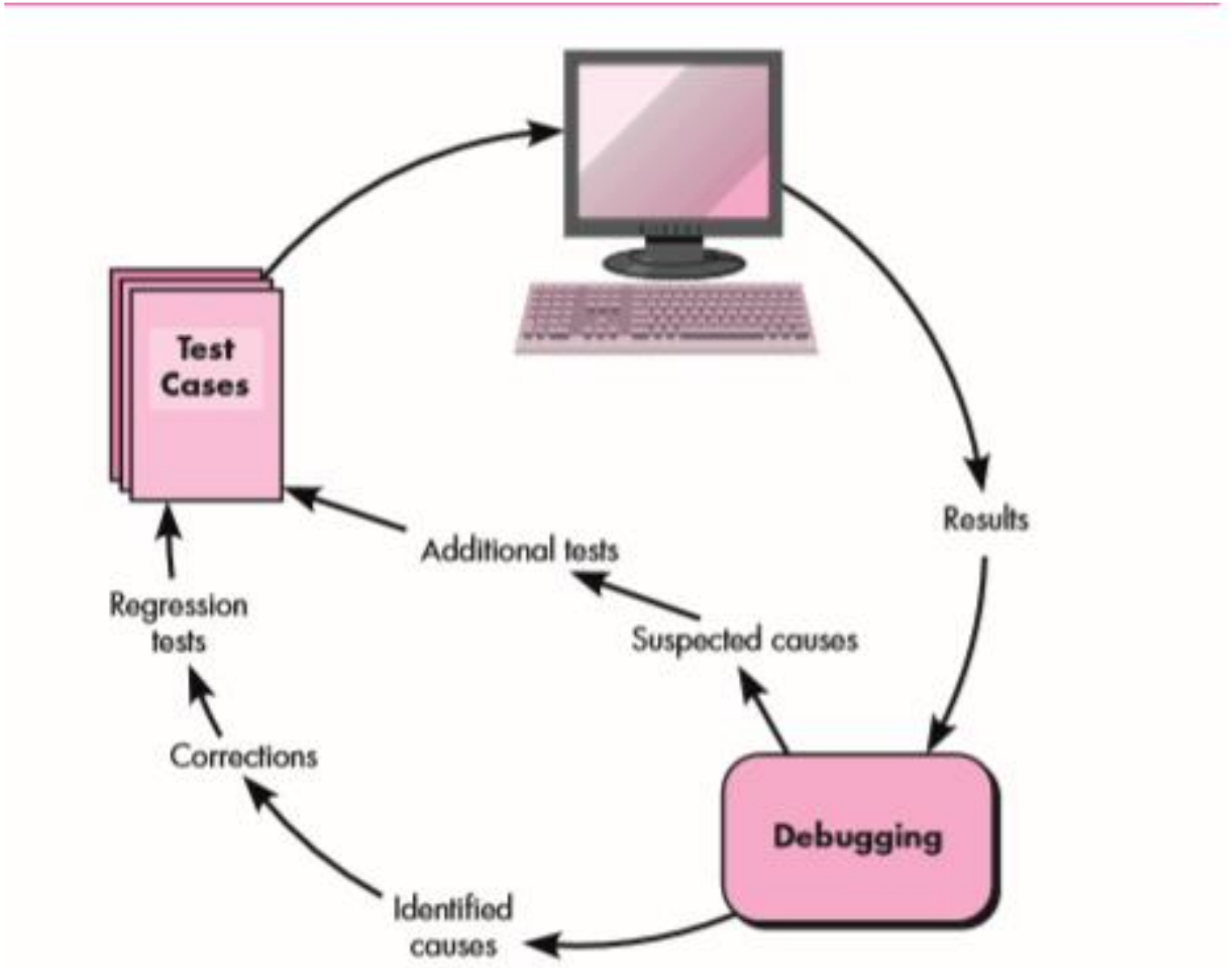
- **Types of System Testing:**
  - **Recovery testing:** how well and quickly does the system recover from faults.
  - **Security testing**: verify that protection mechanisms built into the system will protect from unauthorized access (hackers, disgruntled employees, fraudsters).
  - **Stress testing:** place abnormal load on the system.
  - **Performance testing:** investigate the run-time performance within the context of an integrated system .

The Art of Debugging

- Debugging is not testing but often occurs as a consequence of testing.
- The debugging process begins with the execution of a test case.
- 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.

  (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.


Test Cases
Results
Additional tests
Regression tests
Suspected causes
Corrections
Debugging
Identified causes

Software Testing Fundamentals

- 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 Bach1 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.

Software Testing Fundamentals

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.

Observability: “What you see is what you test.” Inputs provided as part of testing produce distinct outputs. System states and variables are visible or quarriable 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 hardware states and variables can be controlled directly by the test engineer.

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.

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.

White Box Testing

- White box testing is also called as glass-box testing.
- Using white-box testing methods can derive test cases that

– guarantee that all independent paths within a module have been exercised at least once.

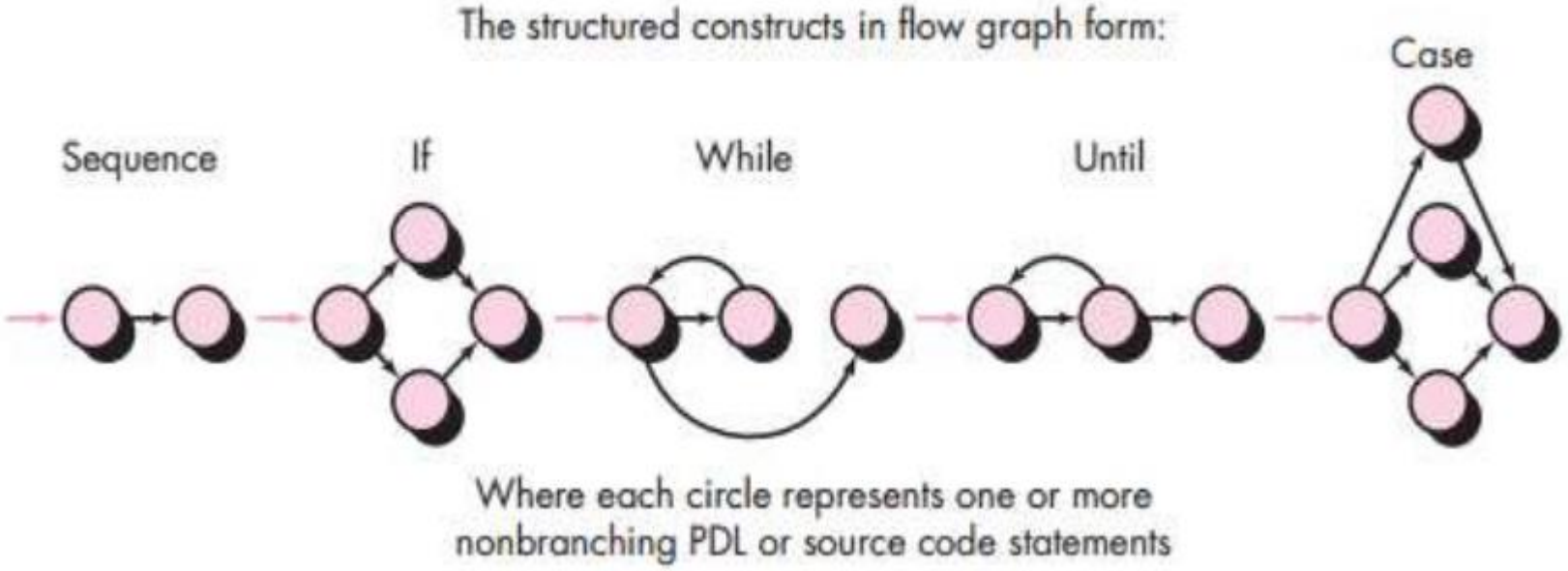
– exercise all logical decisions on their true and false sides.

– execute all loops at their boundaries and within their operational bounds.

– exercise internal data structures to ensure their validity.

Basis Path Testing

- Basis path testing is a white-box testing technique.
- 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.
- Flow Graph Notation: a simple notation for the representation of control flow, called a flow graph. It also know as program graph.



- Arrows called edges or links represent flow of control.
- Circles called floe graph nodes represent one or more actions.
- Areas bounded by edges and nodes called regions.
- A predicate node is a node containing a condition.

- Independent program paths:

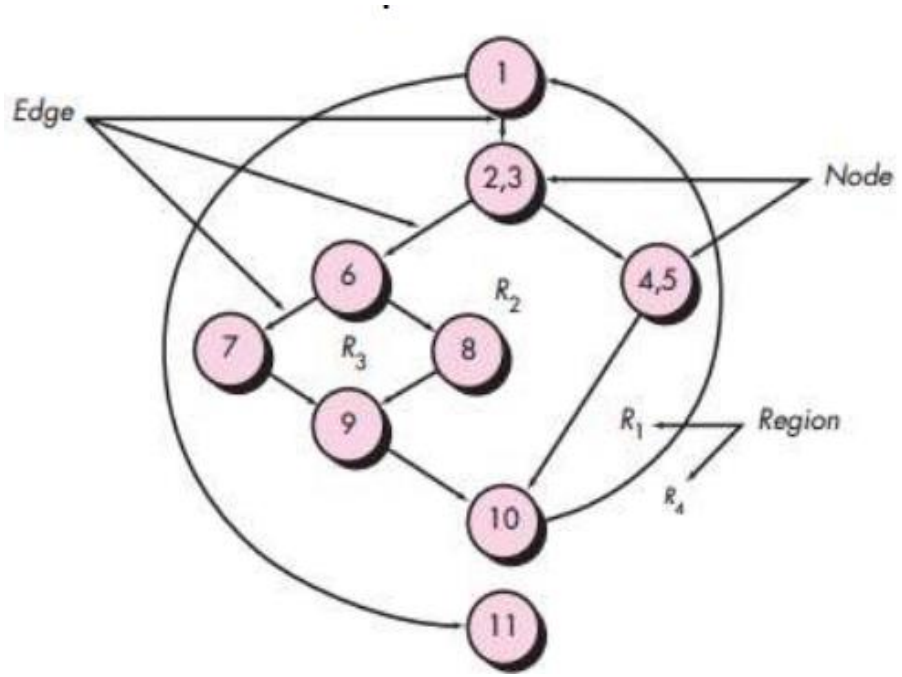
– An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

– independent path must move along at least one edge that has not been traversed before the path is defined.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

– independent path must move along at least one edge that has not been traversed before the path is defined.

– Example:



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

- **Deriving test cases:**

  – Using the design or code as a foundation, draw a corresponding flow graph.

  – Determine the cyclomatic complexity of the resultant flow graph. – Determine a basis set of linearly independent paths.

  – Prepare test cases that will force execution of each path in the basis set.

- **Graph matrices:**

  – 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.

Control Structure Testing

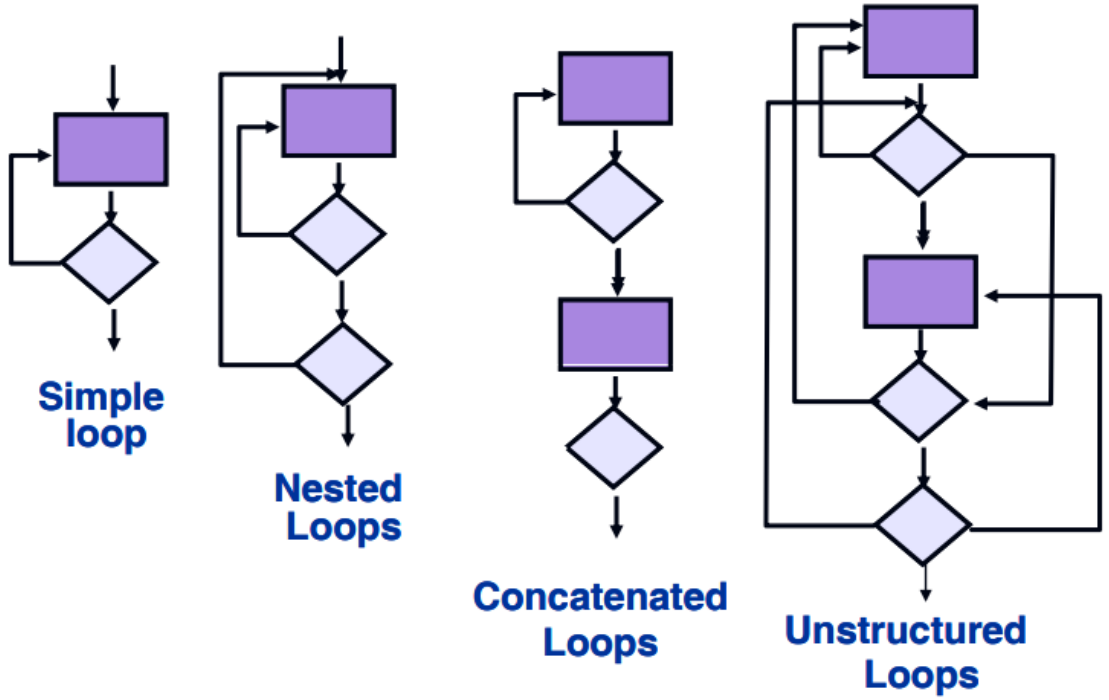
- **Condition Testing:**

A test case design method that exercises the logical conditions contained in a program module.

- **Data Flow Testing:**

Selects test paths of a program according to the locations of definitions and uses of variables in the program.

- **Loop Testing:**



Techniques for Black Box Testing

Sameera Abu Ghalyoun

PPU

What is Black Box Testing?

Black box testing refers to a software testing method where the SUT (Software Under Test) functionality is tested without worrying about its details of implementation, internal path knowledge and internal code structure of the software

Techniques of Black Box Testing

Equivalence Partitioning

- *This technique is also known as Equivalence Class Partitioning (ECP).*
- *In this technique, input values to the system or application are divided into different classes or groups based on its similarity in the outcome.*
- *Hence, instead of using each and every input value we can now use any one value from the group/class to test the outcome. In this way, we can maintain the test coverage while we can reduce a lot of rework and most importantly the time spent.*

Boundary Value Analysis (BVA)

- *BVA helps in testing any software having a boundary or extreme values.*
- *BVA is capable of identifying the flaws of the limits of the input values rather than focusing on the range of input value.*
- *BVA deals with the edge or extreme output values.*

State Transition Testing

- *This technique usually considers the state, outputs and inputs of a system during a specific period.*
- *It checks for the behavioural changes of a system in a particular state or another state while maintaining the same inputs.*
- *The test cases for this Black box testing technique are created by checking the sequence of transitions and state or events among the inputs.*

Graph-Based Testing

- *Graph based testing involves a graph drawing that depicts the link between the causes (inputs) and the effects (output), which trigger the effects.*
- *This testing utilizes different combinations of output and inputs.*
- *It is a helpful techniques to understand the software's functional performance, as it visualizes the flow of inputs and outputs in a lively fashion.*

Error Guessing Technique

- *This is a classic example of experience based testing.*
- *In this technique, the tester can use his/her experience about the application behaviour and functionalities to guess the error-prone areas. Many defects can be found using error guessing where most of the developers usually make mistakes.*
- ***Few common mistakes that developers usually forget to handle:***
  - *Divide by zero.*
  - *Handling null values in text fields.*
  - *Accepting Submit button without any value.*
  - *File upload without attachment.*
  - *File upload with less than or more than the limit size.*

Comparison Testing

Different independent versions of same software are used to compare to each other for testing in this method.

THANK YOU