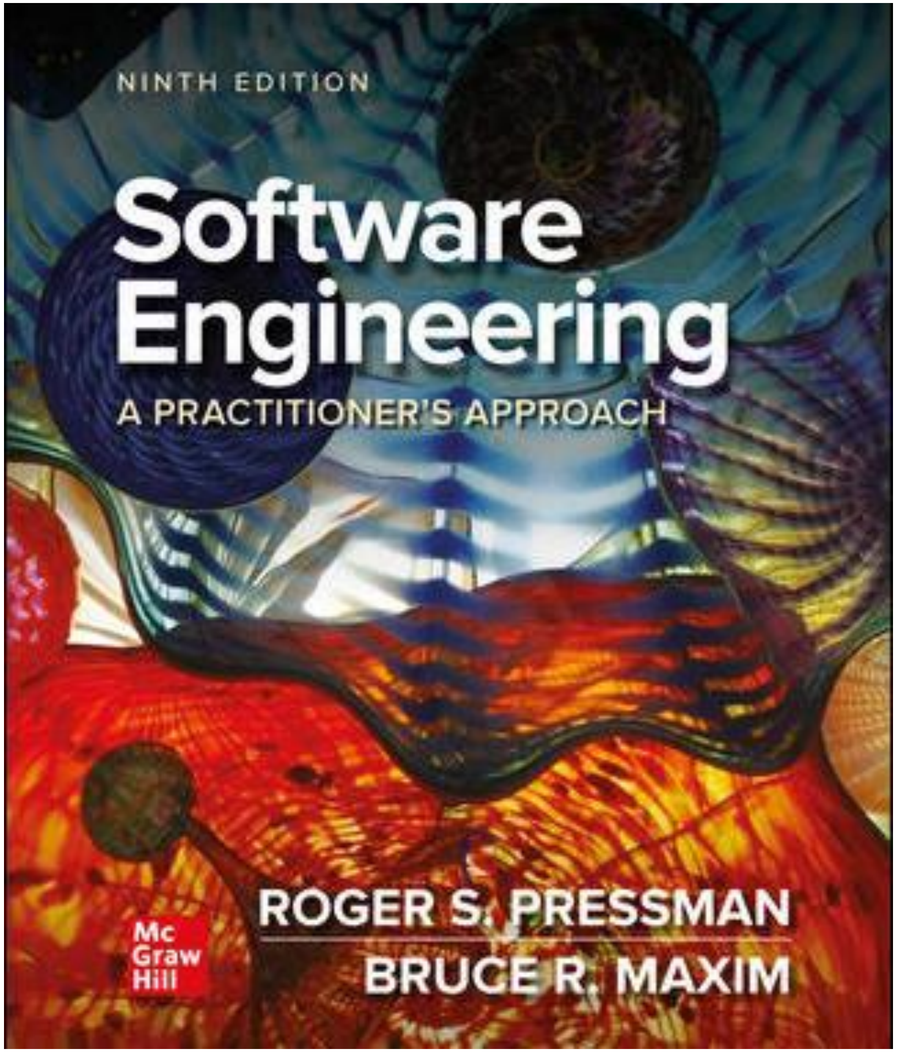
Because learning changes everything.®

Chapter 19

Software Testing – Component Level

Part Three – Quality and Security



Strategic Approach to Testing

- You should conduct effective technical reviews this can eliminate many errors before testing begins.
- Testing begins at the component level and works "outward" toward the integration of the entire 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.

Verification and Validation

Verification refers to the set of tasks that ensure that software correctly implements a specific function.

Verification: Are we building the product right?

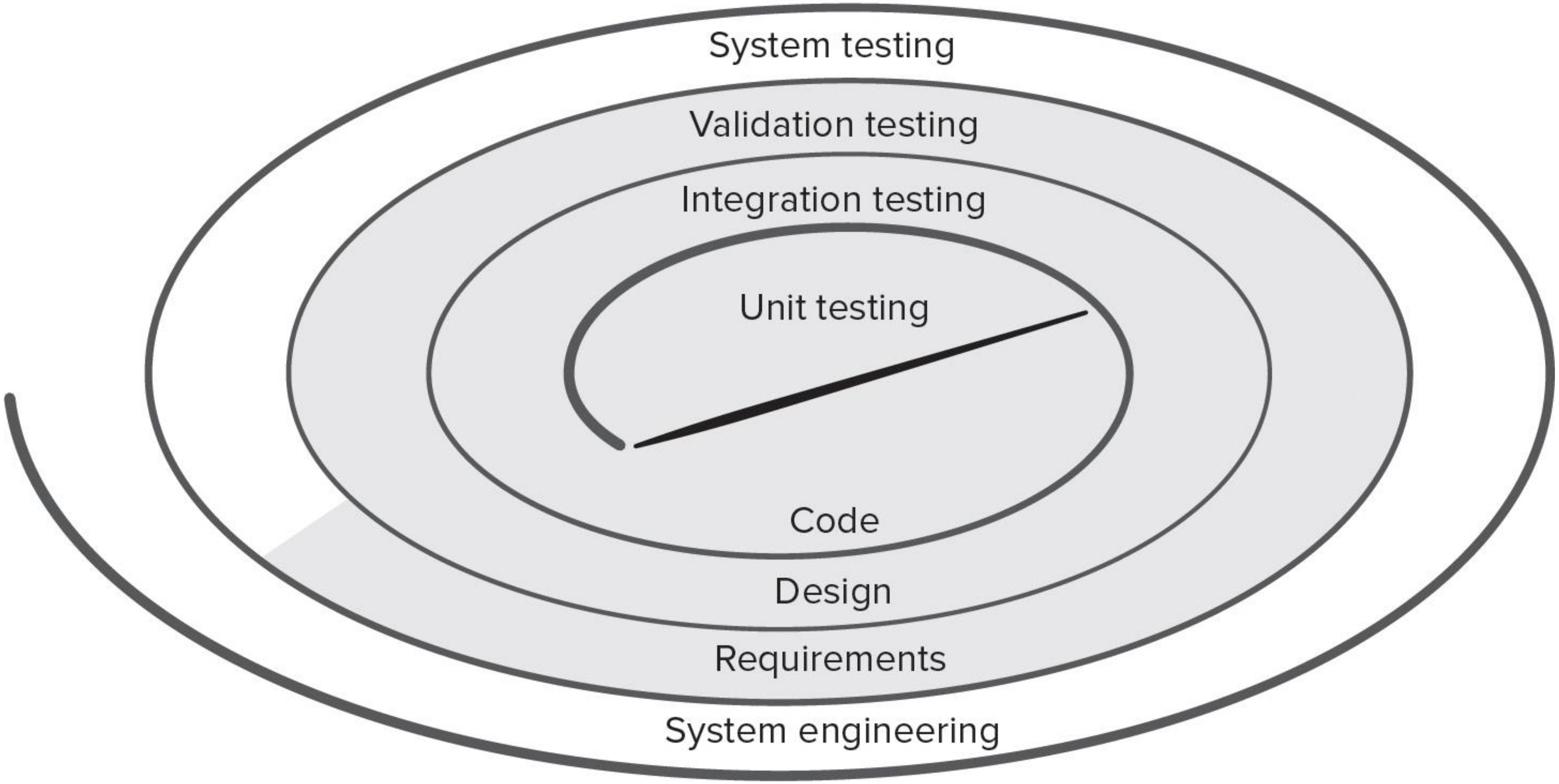
Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Validation: "Are we building the right product?"

Organizing for Testing

- Software developers are always responsible for testing individual program components and ensuring that each performs its deigned function or behavior.
- Only after the 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.
- ITG personnel are paid to find errors.
- Developers and ITG work closely throughout a software project to ensure that thorough tests will be conducted.

Testing Strategy

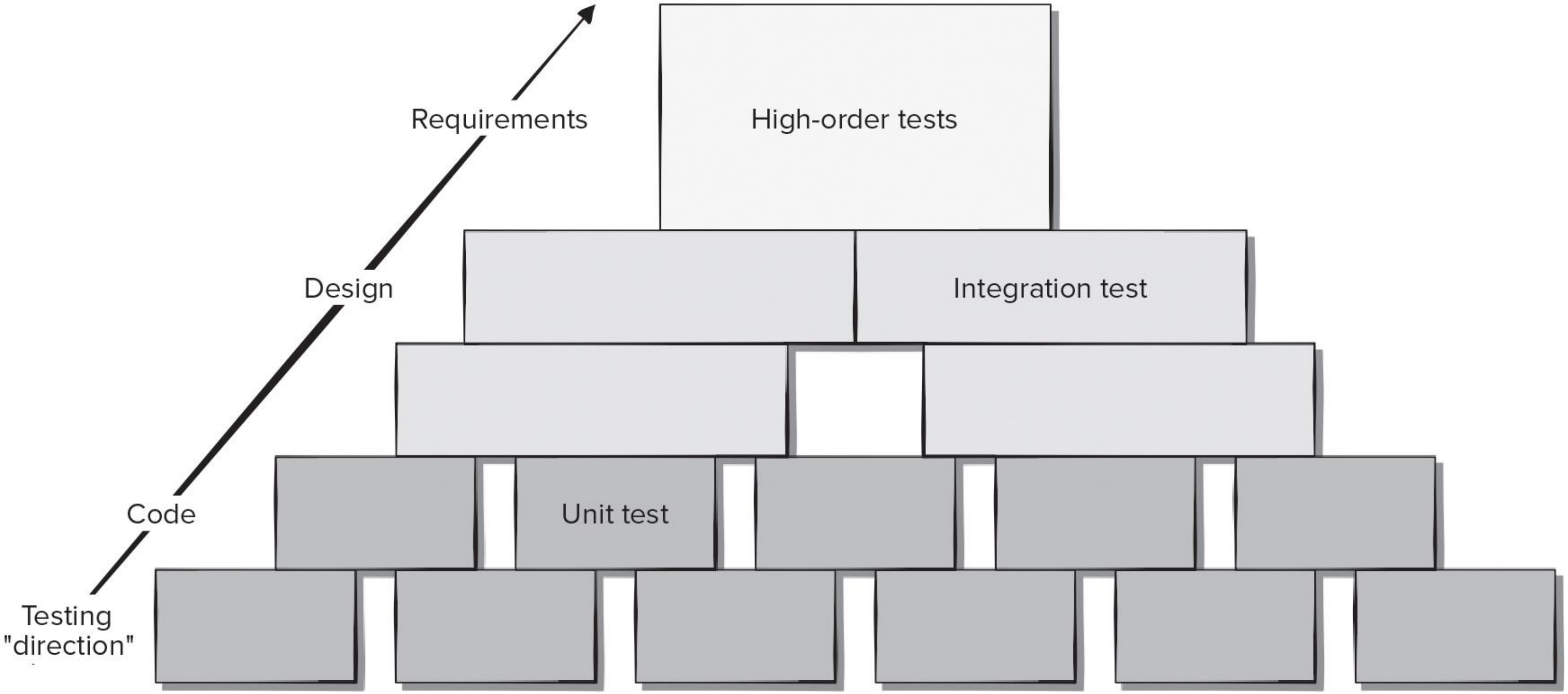


Access the text alternative for slide images.

Testing the Big Picture

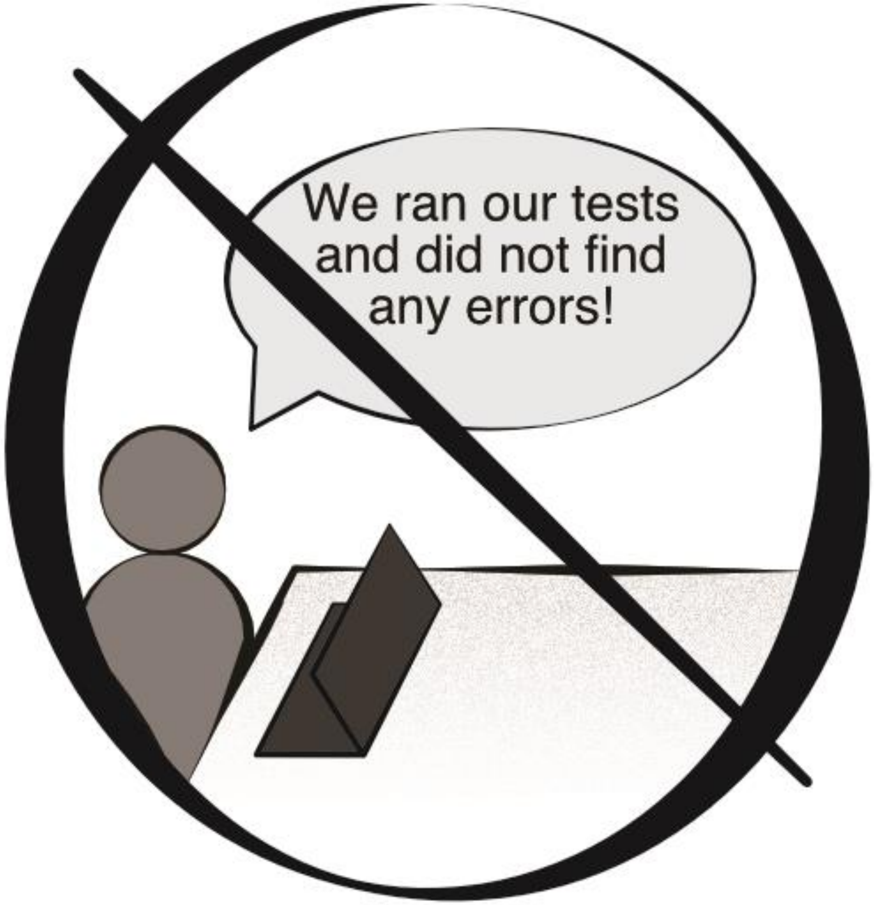
- ***Unit testing*** begins at the center of the spiral and concentrates on each unit (for example, component, class, or content object) as they are implemented in source code.
- Testing progresses to ***integration testing***, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral.
- ***Validation testing***, is where requirements established as part of requirements modeling are validated against the software that has been constructed.
- In ***system testing***, the software and other system elements are tested as a whole.

Software Testing Steps



Access the text alternative for slide images.

When is Testing Done?



Criteria for Done

- You’re never done testing; the burden simply shifts from the software engineer to the end user. (Wrong).
- You’re done testing when you run out of time or you run out of money. (Wrong).
- The ***statistical quality assurance*** approach suggests executing tests derived from a statistical sample of all possible program executions by all targeted users.
- 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?”

Test Planning

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.”
5. Build “robust” software that is designed to test itself.
6. Use effective technical reviews as a filter prior to testing.
7. Conduct technical reviews to assess the test strategy and test cases themselves.
8. Develop a continuous improvement approach for the testing process.

Test Recordkeeping

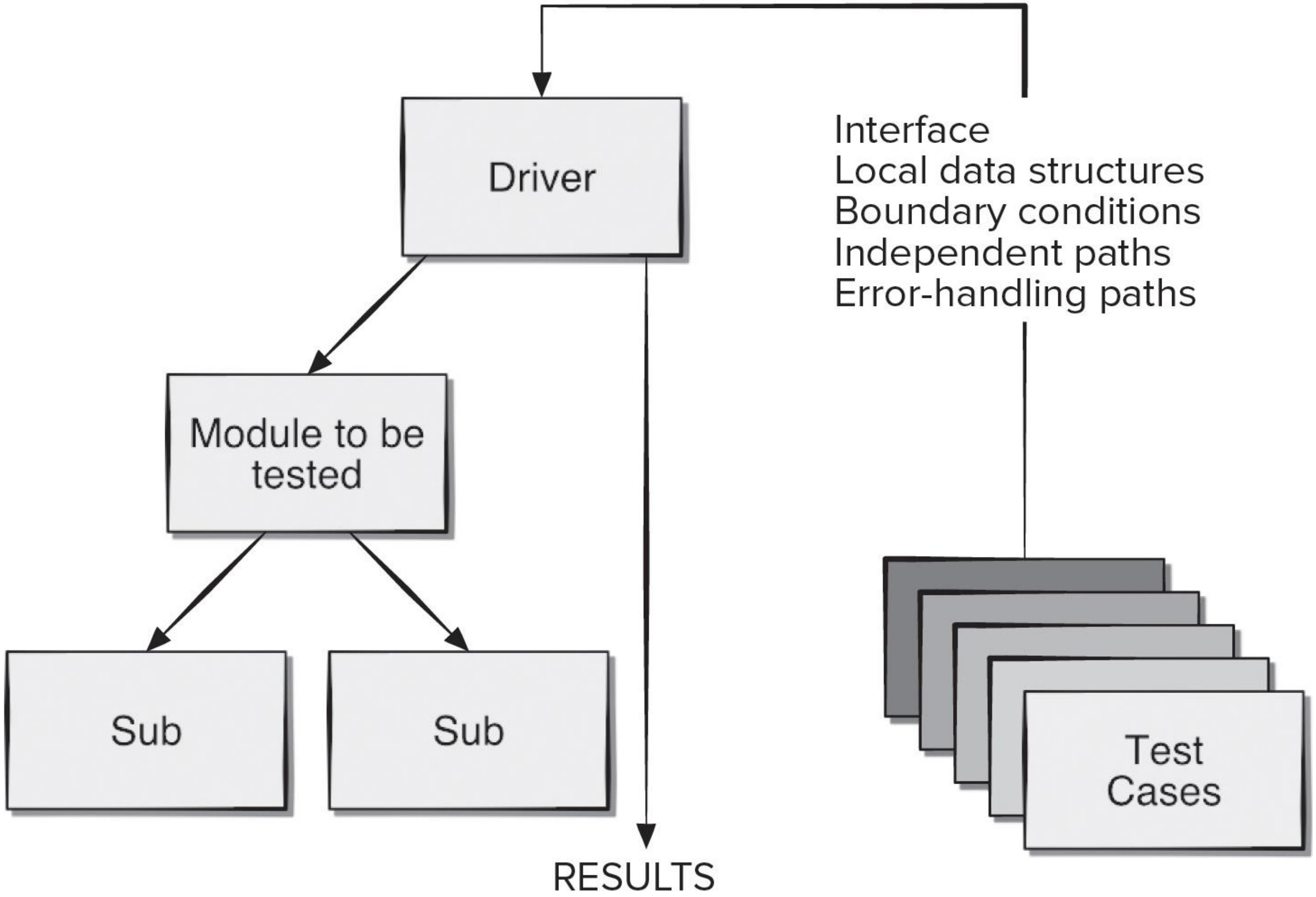
Test cases can be recorded in Google Docs spreadsheet:

- Briefly describes the test case.
- Contains a pointer to the requirement being tested.
- Contains expected output from the test case data or the criteria for success.
- Indicate whether the test was passed or failed.
- Dates the test case was run.
- Should have room for comments about why a test may have failed (aids in debugging).

Role of Scaffolding

- Components are not stand-alone program some type of ***scaffolding*** is required to create a testing framework.
- As part of this framework, driver and/or stub software must often be developed for each unit test.
- 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*** (dummy subprogram) serve to replace modules invoked by the component to be tested.
- A stub uses the module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Unit Test Environment



Access the text alternative for slide images.

Cost Effective Testing

- Exhaustive testing requires every possible combination and ordering of input values be processed by the test component.
- The return on exhaustive testing is often not worth the effort, since testing alone cannot be used to prove a component is correctly implemented.
- Testers should work smarter and allocate their testing resources on modules crucial to the success of the project or those that are suspected to be error-prone as the focus of their unit testing.

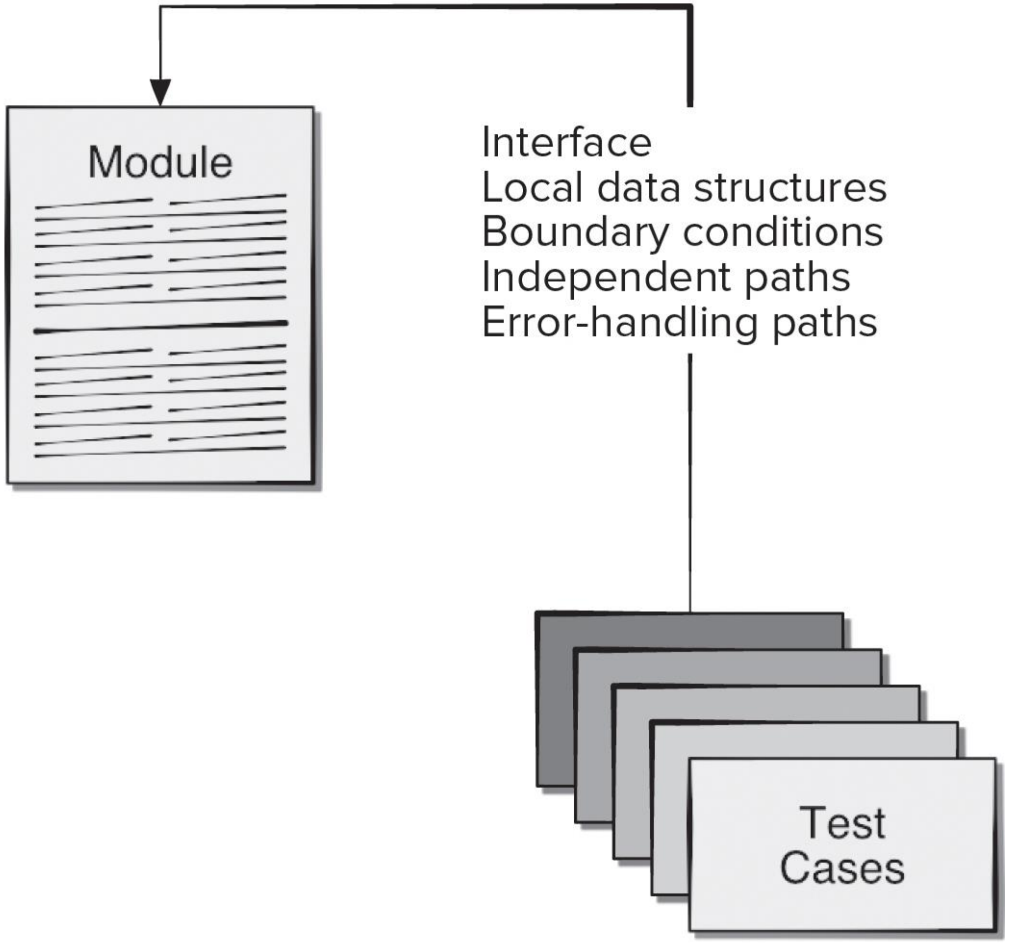
Test Case Design

Design unit test cases before you develop code for a component to ensure that code that will pass the tests.

Test cases are designed to cover the following areas:

- The module interface is tested to ensure that information properly flows into and out of the program unit.
- Local data structures are examined to ensure that stored data stored maintains its integrity during execution.
- Independent paths through control structures are exercised to ensure all statements are executed at least once.
- Boundary conditions are tested to ensure module operates properly at boundaries established to limit or restrict processing.
- All error-handling paths are tested.

Module Tests



Access the text alternative for slide images.

Error Handling

- A good design anticipates error conditions and establishes error-handling paths which 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.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Traceability

- To ensure that the testing process is auditable, each test case needs to be traceable back to specific functional or nonfunctional requirements or anti-requirements.
- Often nonfunctional requirements need to be traceable to specific business or architectural requirements.
- Many test process failures can be traced to missing traceability paths, inconsistent test data, or incomplete test coverage.
- Regression testing requires retesting selected components that may be affected by changes made to other collaborating software components.

White Box Testing

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.
4. Exercise internal data structures to ensure their validity.

Basis Path Testing 1

Determine the number of independent paths in the program by computing 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

   $V(G) = E - N + 2$

   E is the number of flow graph edges

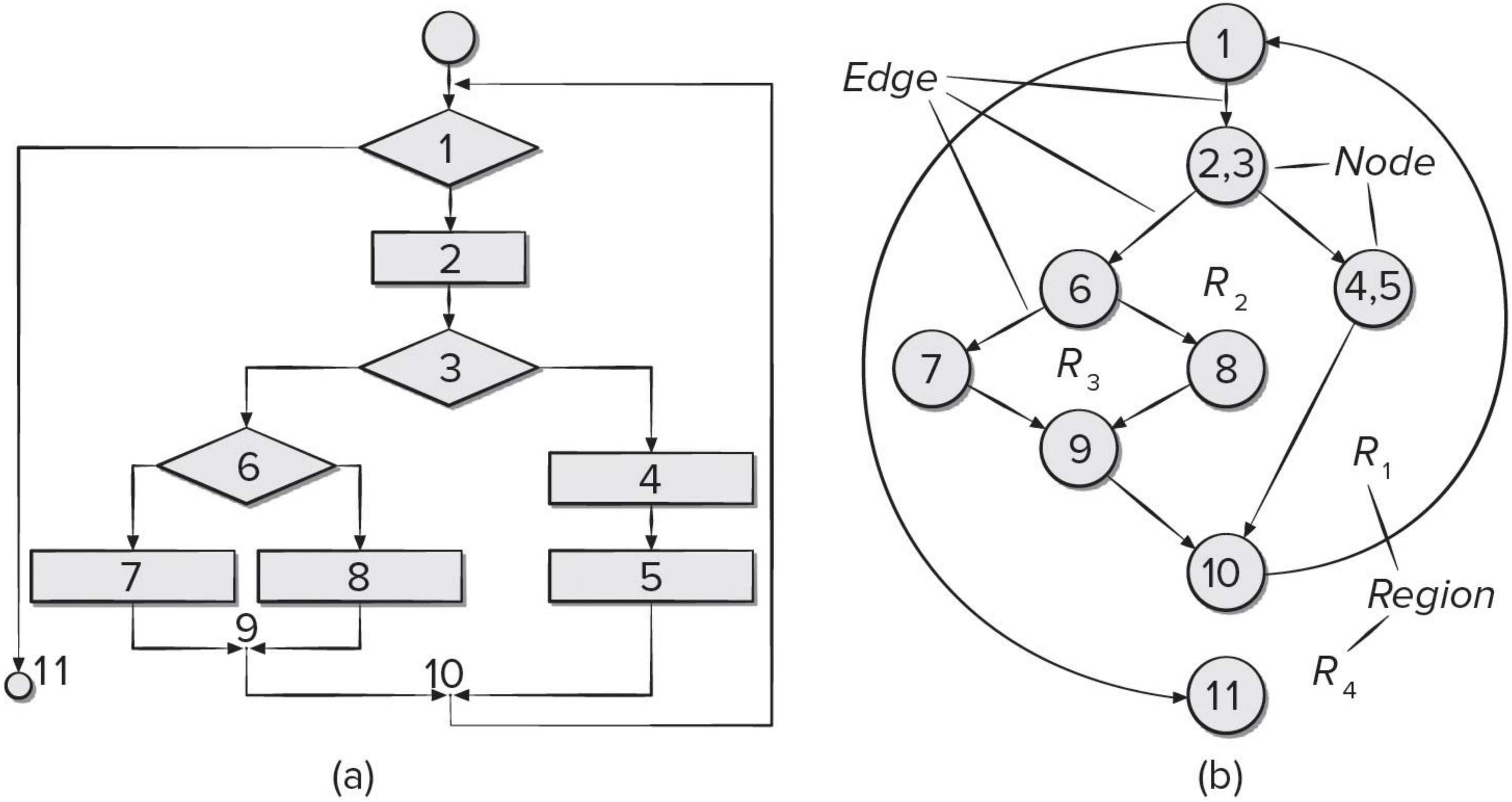
   N is the number of nodes.

3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

   P is number of predicate nodes contained in the flow graph G.

Flowchart (a) and Flow Graph (b)



(a)

(b)

Access the text alternative for slide images.

Basis Path Testing 2

Cyclomatic Complexity of the flow graph is 4

1. The flow graph has four regions.
2. $V(G)$ = 11 edges − 9 nodes + 2 = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

An ***independent path*** is any path through the program that introduces at least one new set of processing statements or a new condition (we need 4 independent paths to test)

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

Basis Path Testing 3

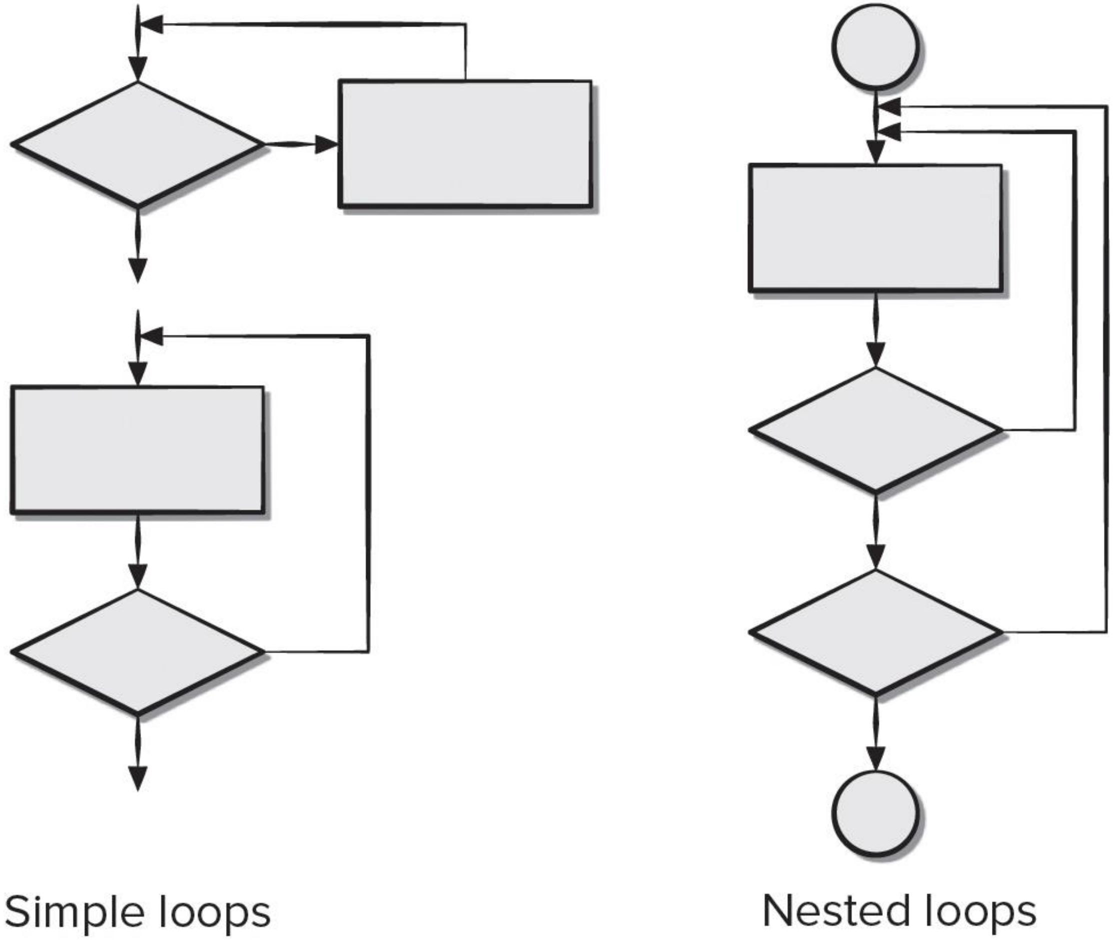
Designing Test Cases

- 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.

Control Structure Testing

- ***Condition testing*** is 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*** is a white-box testing technique that focuses exclusively on the validity of loop constructs.

Classes of Loops



Access the text alternative for slide images.

Loop Testing

Test cases for simple loops:

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.

Test cases for nested loops:

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 (for example, loop counter) values.
3. Add other tests for out-of-range or excluded values.
4. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
5. Continue until all loops have been tested.

Black Box Testing 1

Black-box (functional) 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.
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.

Black Box Testing 2

Black-box test cases are created to answer questions like:

- 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?

Black Box – Interface Testing

- ***Interface testing*** is used to check that a program component accepts information passed to it in the proper order and data types and returns information in proper order and data format.
- Components are not stand-alone programs testing interfaces requires the use stubs and drivers.
- Stubs and drivers sometimes incorporate test cases to be passed to the component or accessed by the component.
- Debugging code may need to be inserted inside the component to check that data passed was received correctly.

Object-Oriented Testing (OOT)

To adequately test OO systems, three things must be done:

- The definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models.
- The strategy for unit and integration testing must change significantly.
- The design of test cases must account for the unique characteristics of OO software.

Black Box – Boundary Value Analysis (BVA)

- ***Boundary value analysis*** leads to a selection of test cases that exercise bounding values.
- Guidelines for BVA:

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 min and max numbers as well as values just above and below min and max.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (for example, array with max index of 100) be certain to design a test case to exercise the data structure at its boundary.

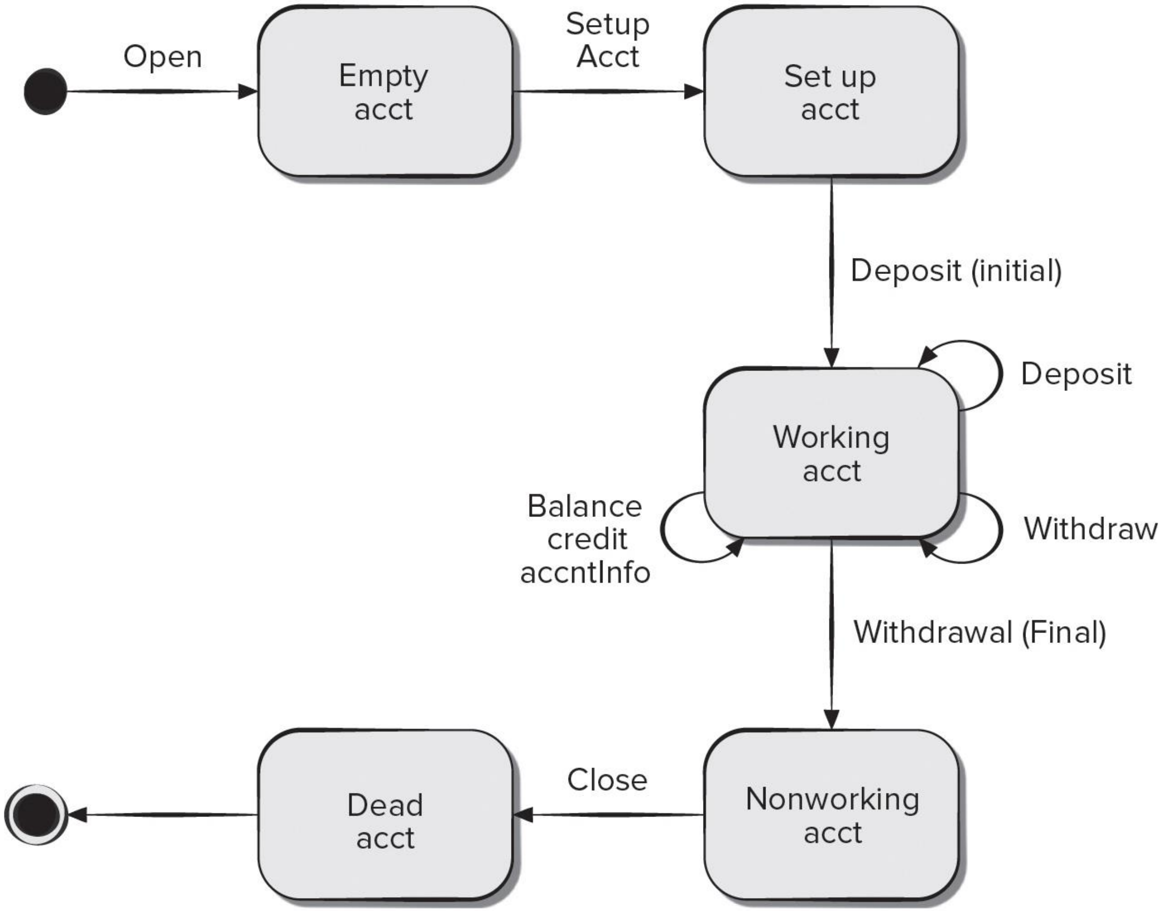
OOT – Class Testing

- Class testing for object-oriented (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.
- Valid sequences of operations and their permutations are used to test that class behaviors - equivalence partitioning can reduce number sequences needed,

OOT– Behavior Testing

- A state diagram can be used to help derive a sequence of tests that will exercise dynamic behavior of the class.
- Tests to be designed should achieve full coverage by using operation sequences cause transitions through all allowable states.
- When class behavior results in a collaboration with several classes, multiple state diagrams can be used to track system behavioral flow.
- A state model can be traversed in a breadth-first manner by having test case exercise a single transition and when a new transition is to be tested only previously tested transitions are used.

State Diagram for Account Class



Source: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 1994, 79.

Access the text alternative for slide images.

Mc
Graw
Hill
Education

Because learning changes everything.®

www.mheducation.com