Software Engineering

Vinaya Sathyanarayana

Component Testing

Testing of Software Components

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?

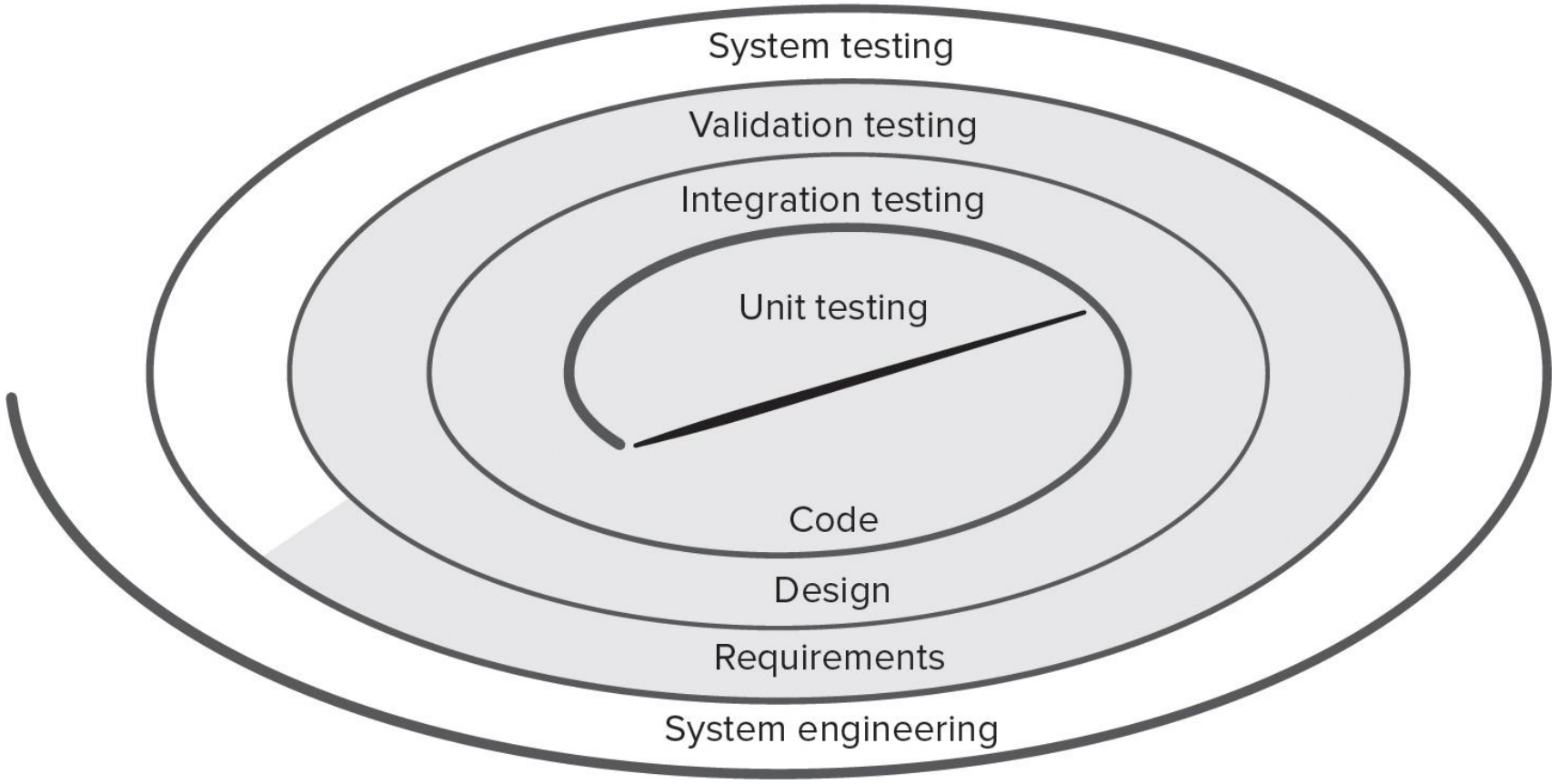
- ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

  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

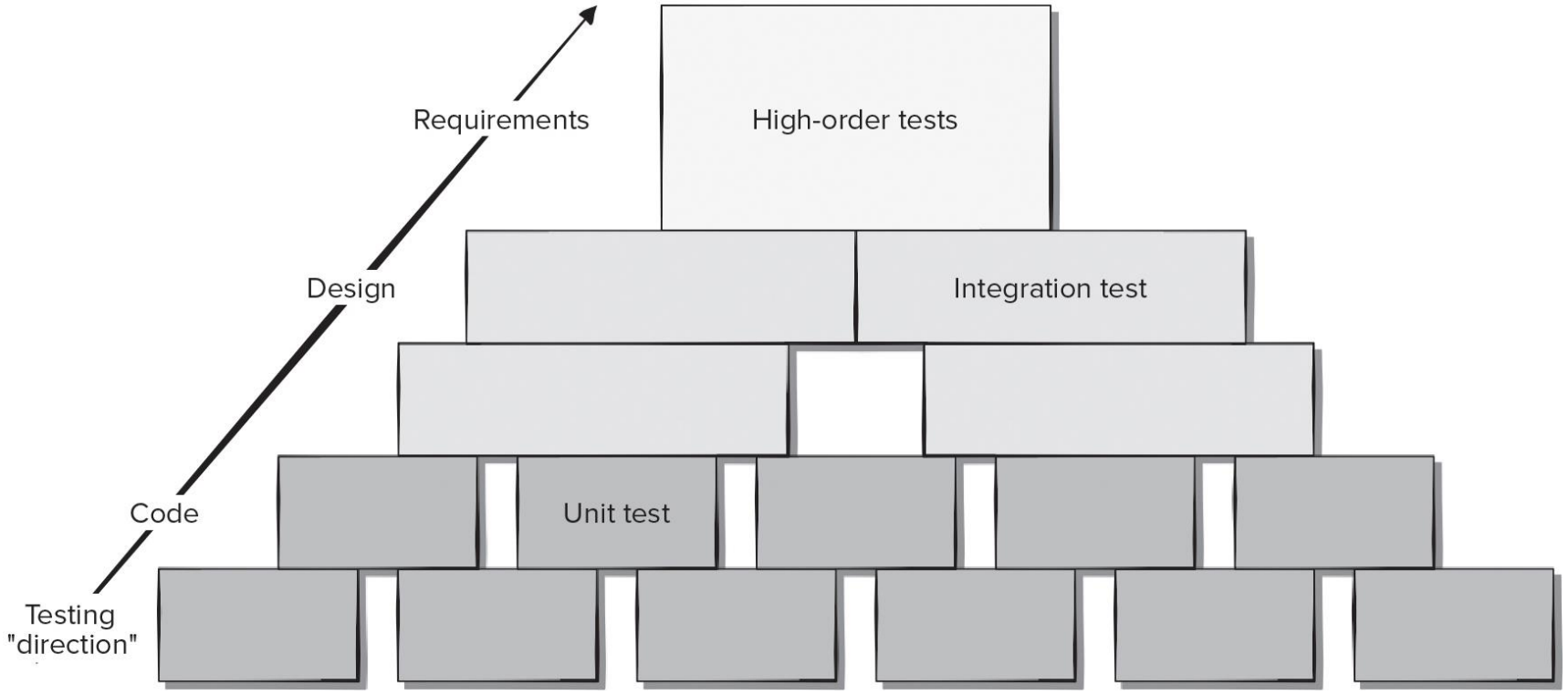


- Access the text alternative for slide images.

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.

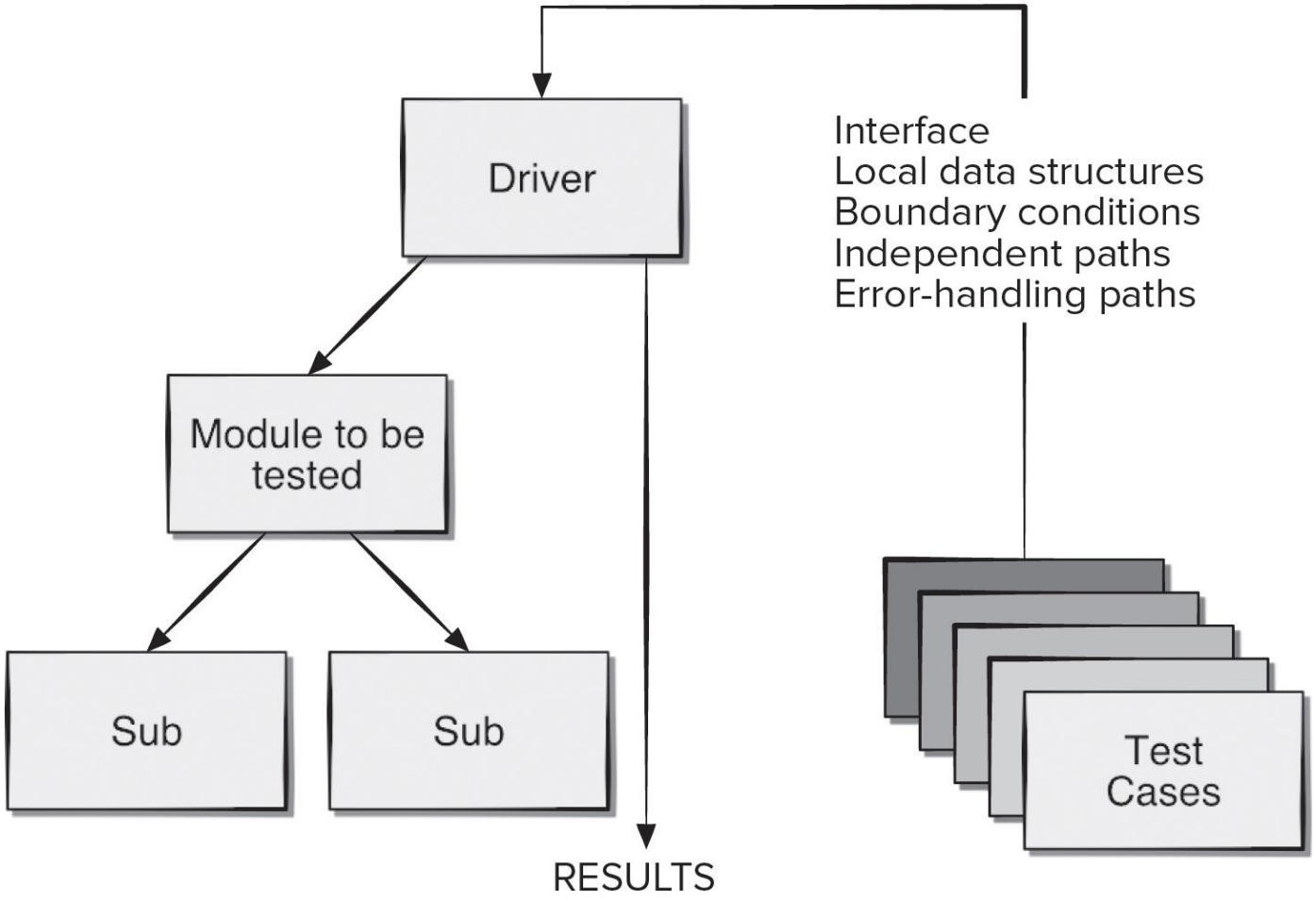
Test Recordkeeping

- 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

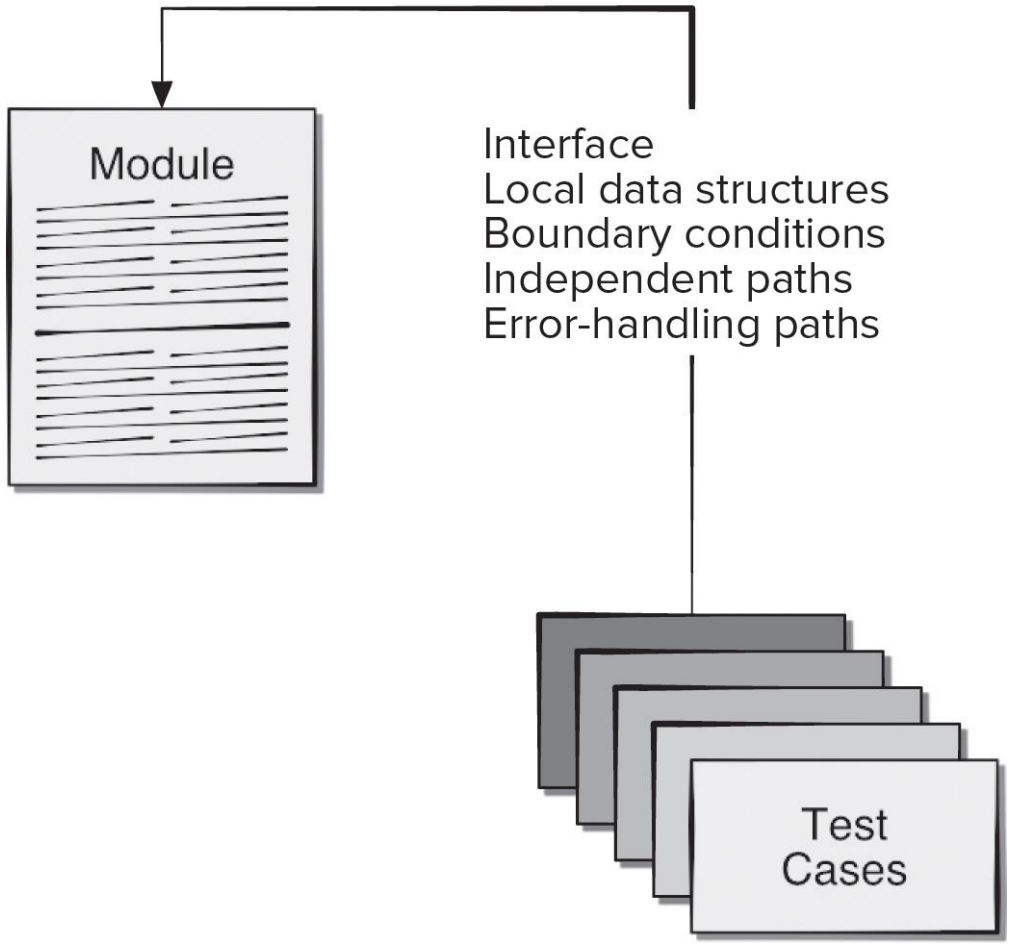
- Exhaustive testing requires every possible combination and ordering of input values be processed by the test component.
- The return on exhaustive testing is often not worth the effort, since testing alone cannot be used to prove a component is correctly implemented.
- Testers should work smarter and 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.

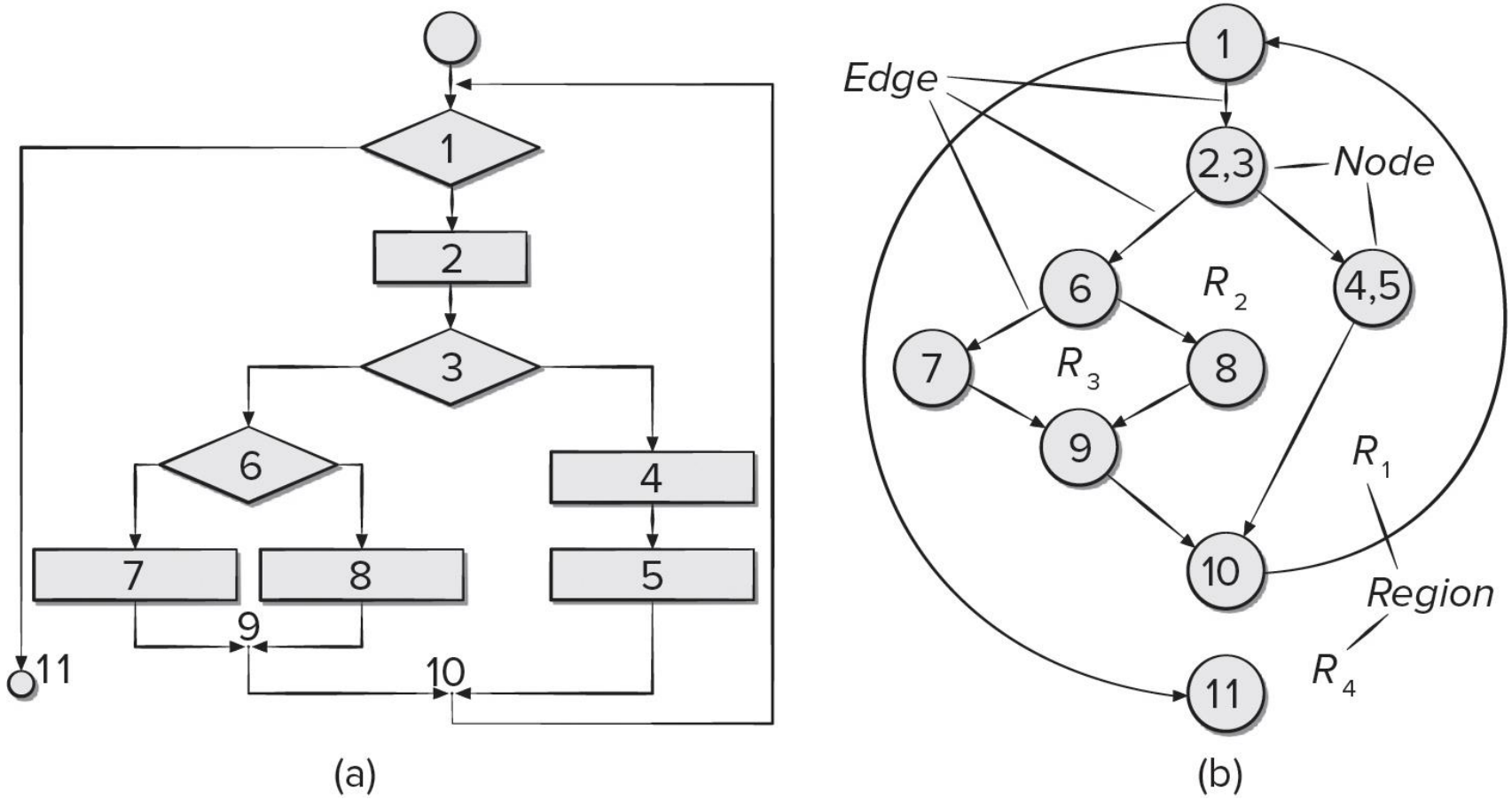
Basis Path Testing [1]

- 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)



- 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

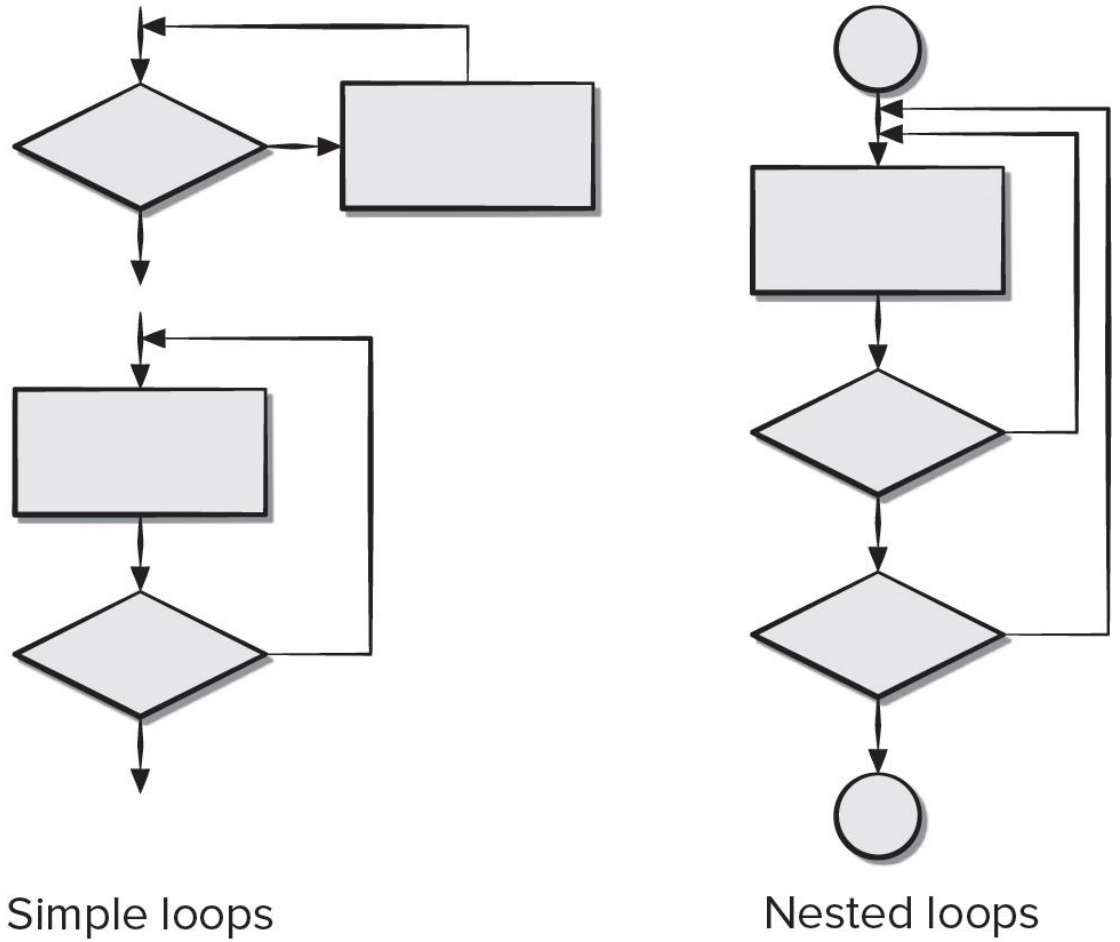
Designing Test Cases

- 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



Simple loops

Nested 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.

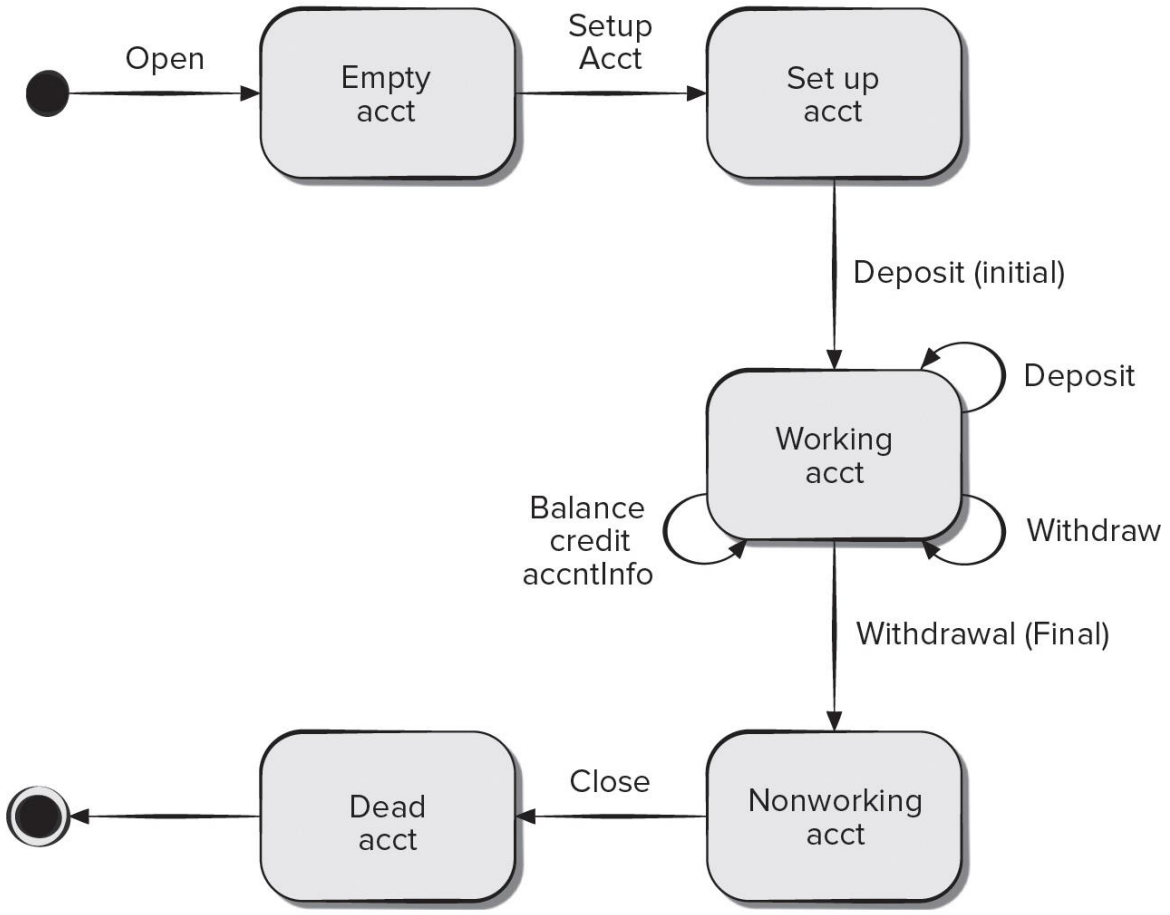
OOT – Class Testing

- 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.

Integration Testing

Testing Fundamentals

- Attributes of a good test:
- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be “best of breed.”
- A good test should be neither too simple nor too complex.

Approaches to Testing

- Any engineered product 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 theinternal 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.

White Box Integration Testing

- ***White-box testing***, is an integration testing philosophy that uses implementation knowledge of the control structures described as part of component-level design to derive test cases.
- White-box tests can be only be designed after source code exists and program logic details are known.
- Logical paths through the software and collaborations between components are the focus of white-box integration testing.
- Important data structures should also be tested for validity after component integration.

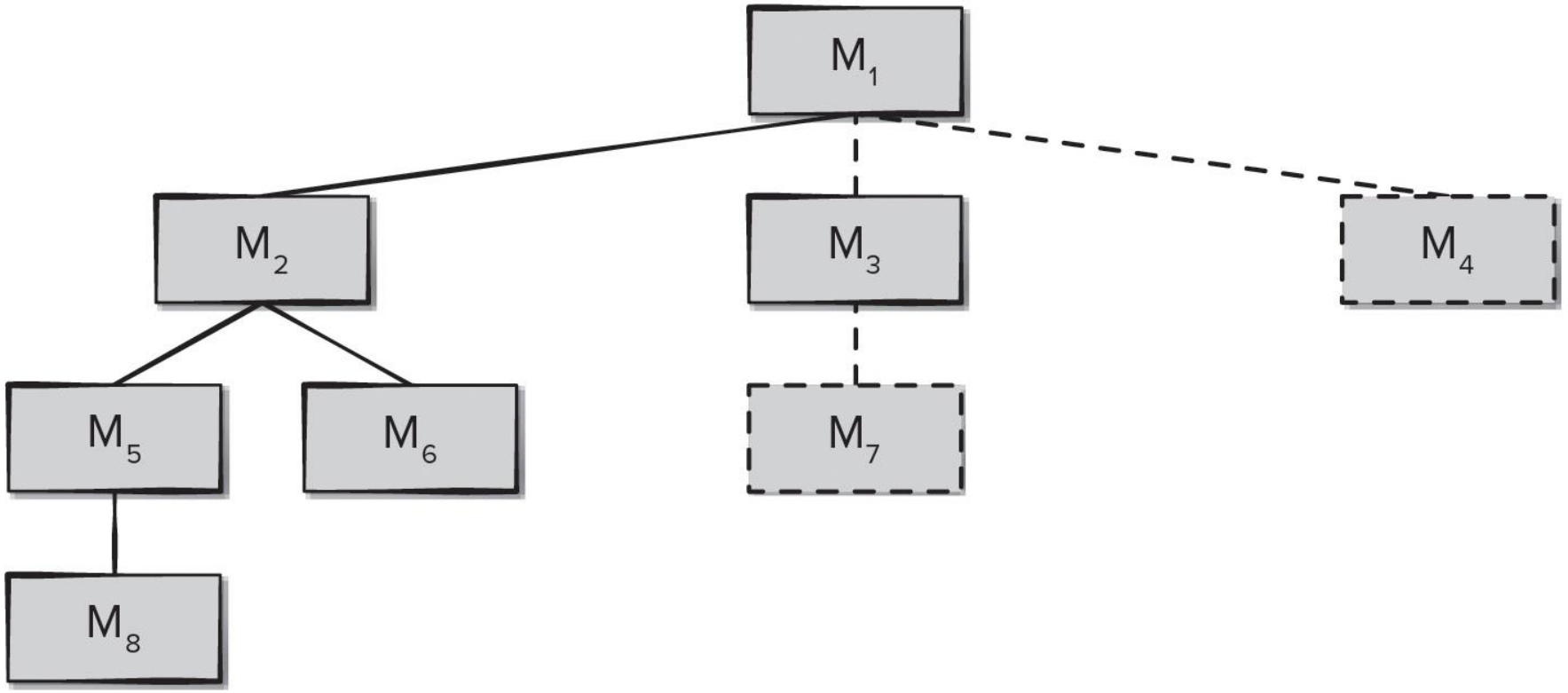
Integration Testing

- ***Integration testing*** is a systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that matches the design.
- In the ***big bang*** approach, all components are combined at once and the entire program is tested as a whole. Chaos usually results!
- In ***incremental integration*** a program is constructed and tested in small increments, making errors easier to isolate and correct. Far more cost-effective!

Top-Down Integration 1

- ***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 to the main control module are incorporated into the structure followed by their subordinates.
- ***Depth-first integration*** integrates all components on a major control path of the program structure before starting another major control path.
- ***Breadth-first integration*** incorporates all components directly subordinate at each level, moving across the

Top-Down Integration 2



- Access the text alternative for slide images.

Top-Down Integration Testing

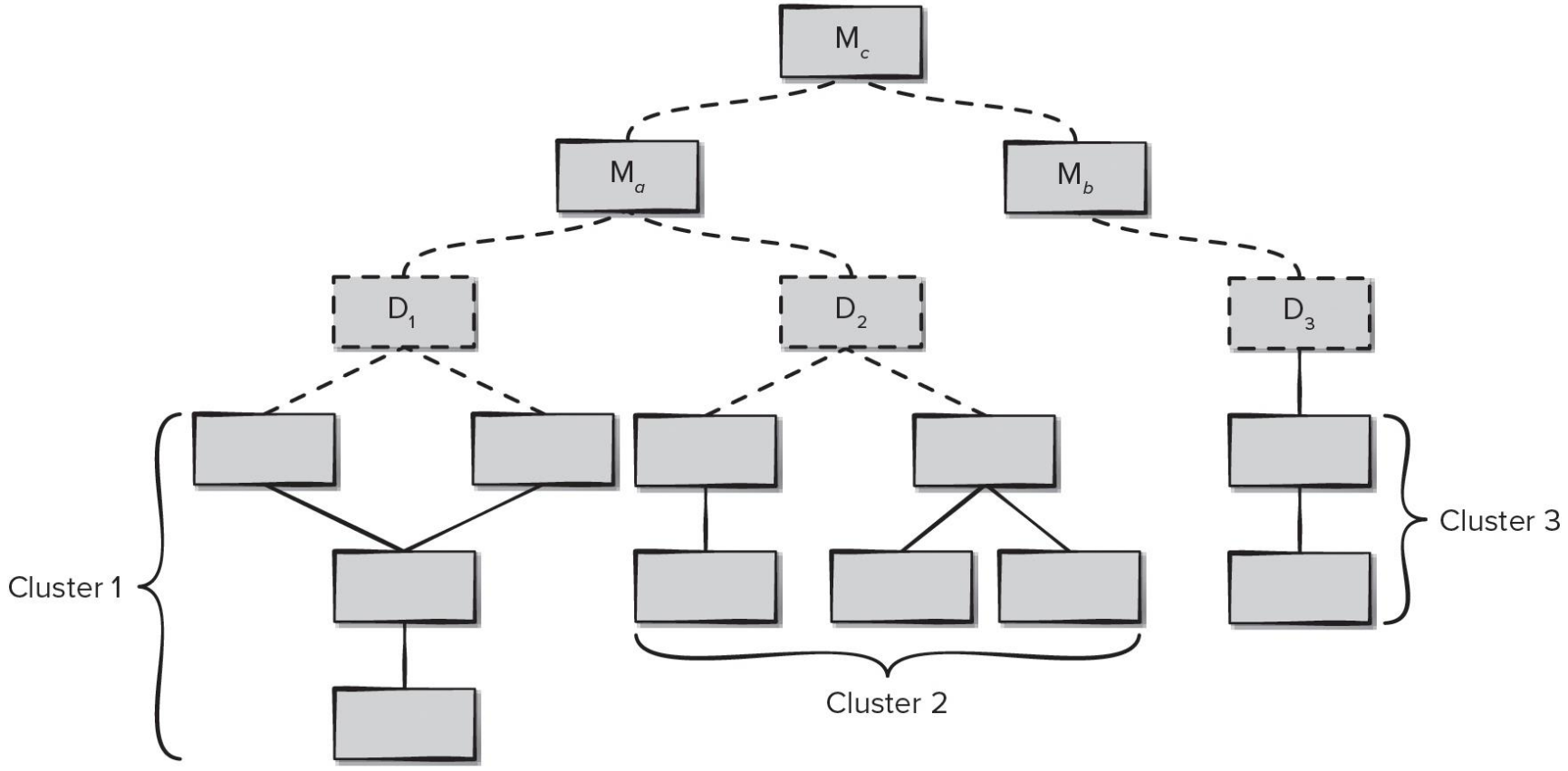
1. The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (for example, depth or breadth first), subordinate stubs are replaced one at a time with actual components.
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.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

Bottom-Up Integration Testing

- ***Bottom-up integration testing***, begins construction and testing with *atomic modules* components at the lowest levels in the program structure.

1. Low-level components are combined into clusters (***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.

Bottom-Up Integration



- Access the text alternative for slide images.

Continuous Integration

- ***Continuous integratio****n* is the practice of merging components into the evolving software increment at least once a day.
- This is a common practice for teams following agile development practices such as XP or DevOps. Integration testing must take place quickly and efficiently if a team is attempting to always have a working program in place as part of continuous delivery.
- ***Smoke testing*** is an integration testing approach that can be used when software is developed by an agile team using short increment build times.

Smoke Testing Integration

1. Software components that have been translated into code are integrated into a *build*. – that includes all data files, libraries, reusable modules, and components required to implement one or more product functions.
2. A series of tests is designed to expose “show-stopper” errors that will keep the build from properly performing its function cause the project to fall behind schedule.
3. The build is integrated (either top-down or bottom-up) with other builds, and the entire product (in its current form) is smoke tested daily.

Smoke Testing Advantages

- **Integration risk is minimized**, since smoke tests are run daily.
- **Quality of the end product is improved**, functional and architectural problems are uncovered early.
- **Error diagnosis and correction are simplified**, errors are most likely in (or caused by) the new build.
- **Progress is easier to assess**, each day more of the final product is complete.
- Smoke testing resembles regression testing by ensuring newly added components do not interfere with the behaviors of existing components.

Integration Testing Work Products

- An overall plan for integration of the software and a description of specific tests is documented in a ***test specification.***
- Test specification incorporates a test plan and a test procedure and becomes part of the software configuration.
- Testing is divided into phases and incremental builds that address specific functional and behavioral characteristics of the software.
- Time and resources must be allocated to each increment build along with the test cases needed.
- A history of actual test results, problems, or peculiarities is recorded in a ***test report*** and may be

Regression Testing

- ***Regression testing*** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- 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 re-executing a subset of all test cases or using automated capture/playback tools.

OO Integration Testing

- ***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 no side effects occur.

- ***Use-based testing***, begins the construction of the system by testing those classes (called ***independent classes***) that use very few *server* classes.

  The next layer classes, (called ***dependent classes)*** use the independent classes are tested next.

  This sequence of testing layers of dependent classes continues until the entire system is constructed.

OO Testing – Fault-Based Test Case Design

- The object of *fault-based testing* is to design tests that have a high likelihood of uncovering plausible faults.
- Because the product or system must conform to customer requirements, fault-based testing begins with the analysis model.
- The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.
- To determine whether these faults exist, test cases are designed to exercise the design or code.

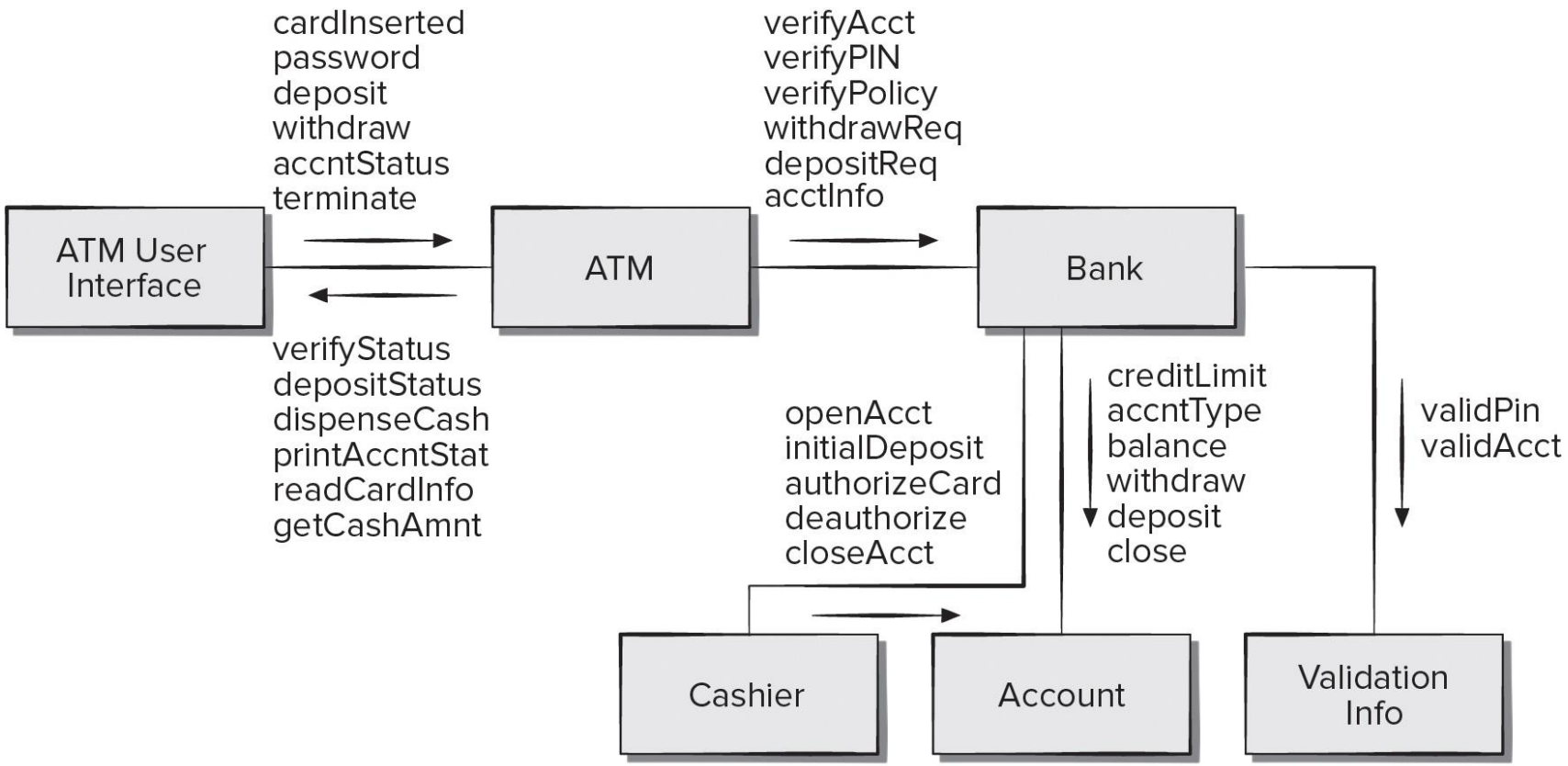
Fault-Based OO Integration Testing

- Fault-based integration testing looks for plausible faults in operation calls or message connections:
  - unexpected result
  - wrong operation/message used
  - incorrect invocation
- Integration testing applies to attributes andoperations – class behaviors are defined by the attributes.
- Focus of integration testing is to determine whether errors exist in the calling (client) code, not the called (server) code.

OO Testing – Fault-Based Test Case Design

- ***Scenario-based testing*** uncovers 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.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes

Collaboration Diagram for Banking Application



- Access the text alternative for slide images.

- Source: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 4, 1994, 72.

OO Testing – Random Test Case Design

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.
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.

- A random test case for the **Bank** class might be
- *Test case r3* = verifyAcct verifyPIN depositReq

OO Testing – Scenario-Based Test Case Design

- ***Scenario-based testing*** uncovers 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.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes

Validation Testing

- ***Validation testing*** tries to uncover errors, but the focus is at the requirements level - on user visible actions and user-recognizable output from the system.
- Validation testing begins at the culmination of integration testing, the software is completely assembled as a package and errors have been corrected.
- Each user story has user-visible attributes, and the customer’s acceptance criteria which forms the basis for the test cases used in validation-testing.
- A ***deficiency list*** is created when a deviation from a specification is uncovered and their resolution is negotiated with all stakeholders.

Software Testing Patterns

- Testing patterns are described in much the same way as design patterns.
- *Example:*

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.

Mobile Testing

Creating a Mobile Test Plan

- 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?
- When is lab testing versus remote testing appropriate.

Mobile Testing Guidelines 1 [1]

- Understand the network landscape and device landscape.
- Conduct testing in uncontrolled real-world test conditions.
- Select the right automation test tool.
- Identify the most critical hardware/ platform combinations to test.

Mobile Testing Guidelines 1 2

- Check the end-to-end functional flow in all possible platforms at least once.
- Conduct performance, GUI, and compatibility testing using actual devices.
- Measure Mobile performance under realistic network load conditions.

Mobile Testing Strategies 1

- **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 (for example, download times, processor speed, storage capacity, power availability).

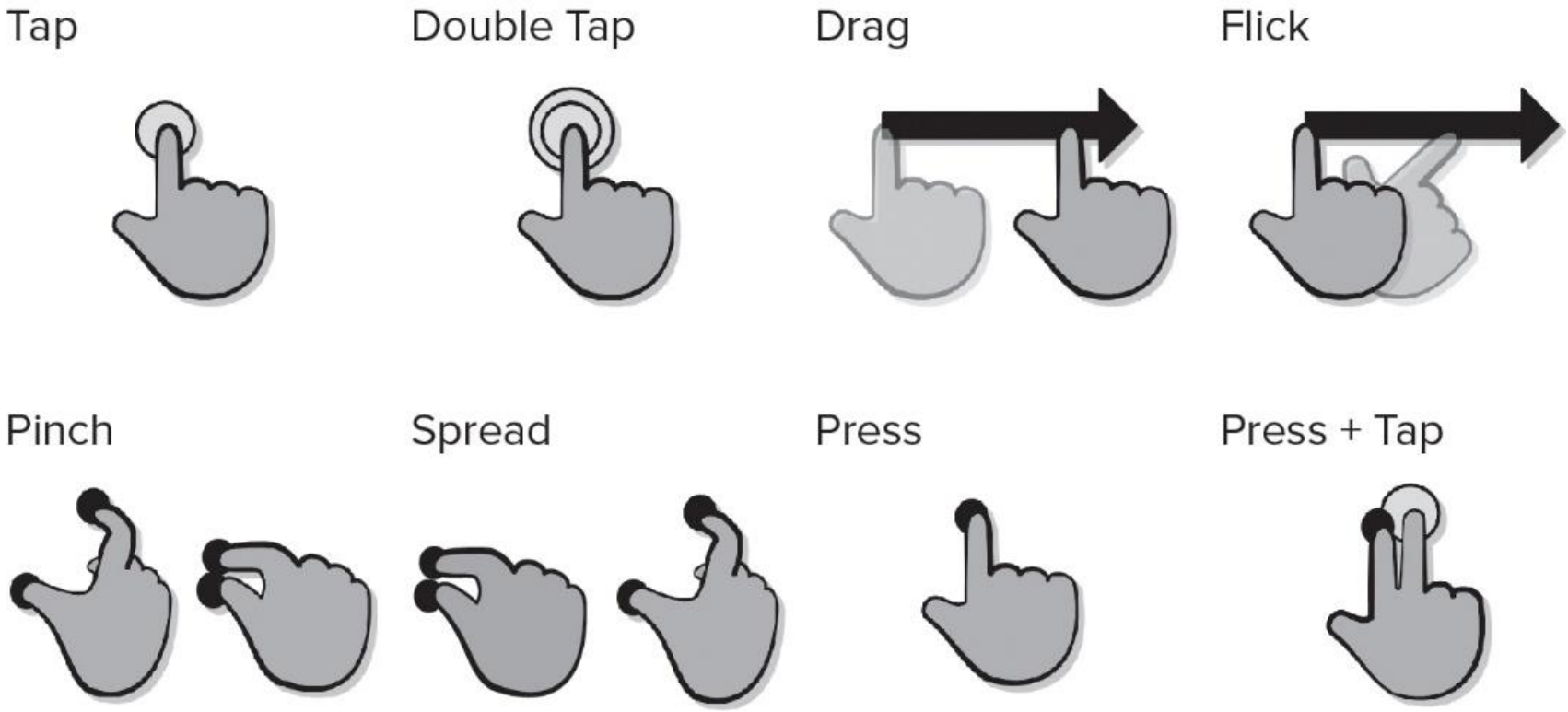
Mobile Testing Strategies 2

- **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.

UX – Gesture Testing Issues

- Touch screens are ubiquitous on mobile devices and developers have added multitouch gestures as a means of augmenting the user interaction possibilities without losing screen real estate.
- Paper prototypes cannot be used to adequately review the adequacy or efficacy of gestures.
- It’s difficult to use automated tools to test gesture interface actions.
- Location of screen objects is affected by screen size and resolution making accurate gesture testing difficult.
- Gestures are hard to log accurately for replay.
- Accessibility testing for visually impaired users is

Add Body Slides in this Portion Presentation [1]



- Access the text alternative for slide images.

UX – Virtual Keyboards

- Virtual keyboard may obscure part of the display screen when activated, it is important to ensure that important screen information is not hidden from the user while typing.
- Virtual keyboards are smaller than personal computer keyboards, it is hard to type with 10 fingers and they 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*** (that is, autocompletion of partially typed words) are often used with to help expedite user input.

UX – Voice Input

- Voice input has become an increasingly common method for providing input and commands in hands-busy, eyes-busy situations.
- Using voice commands to control a device impresses a greater cognitive load on the user, than pointing to a screen object or pressing a key.
- Testing the quality and reliability of voice input and recognition needs to take environmental conditions and individual voice variation into account.
- The MobileApp should be tested to ensure that bad input does not crash the MobileApp or the device.
- It is important to log errors to help developers improve the ability of the MobileApp to process

UX – Alerts and Extraordinary Conditions

- 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,
- When foreign languages are involved, verification that the translation from one language to another is correct.
- You should not rely solely on testing in a development environment and you must test MobileApp in the wild on actual devices.
- Apps must recover from faults and resume processing

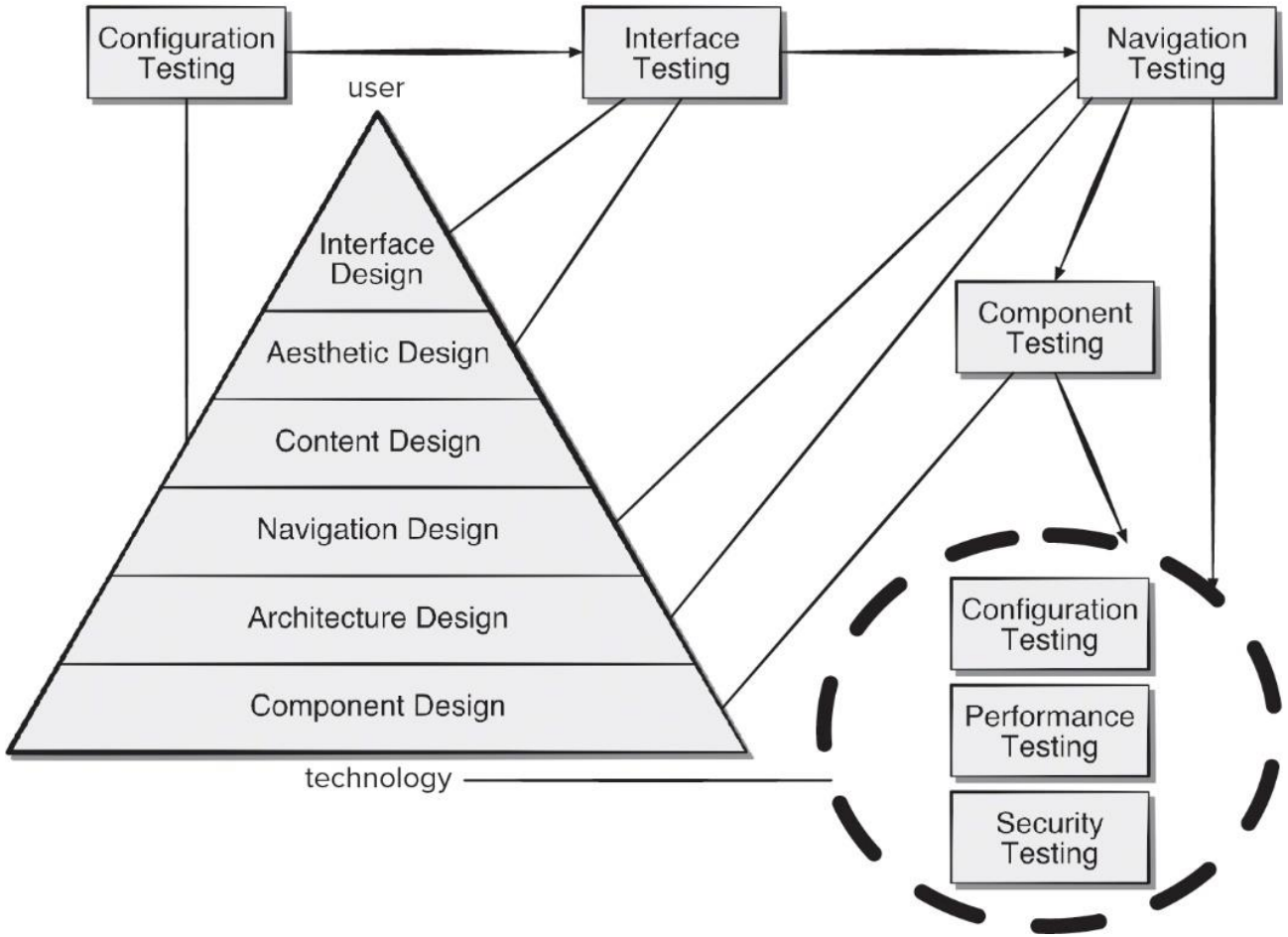
Web App Testing Steps 1 [1]

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.

Web App Testing Steps 1 2

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 errors.

Add Body Slides in this Portion Presentation 2



- Access the text alternative for slide images.

Web App – Content Testing

- Content testing has three important objectives:

1. to uncover syntactic errors (for example, typos, grammar mistakes) in text-based documents, graphical representations, and other media.
2. to uncover semantic errors (that is, 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.

Assessing Content Sematics

- 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 content offensive, misleading, or an open door to litigation?

Web App – Interface Testing

- Interface features tested to ensure design rules, aesthetics, and content is available to user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.
- Each interface mechanism is tested within the context of a use-case or NSU for a specific user category.
- Complete interface is tested against selected use-cases and NSUs to uncover errors in interface semantics.
- The interface is tested within a variety of environments (for example, browsers) to ensure that it will be compatible.

Web App – Navigation Testing 1

- Answer these questions as each NSU or use case 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

Web App – Navigation Testing 2

- 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

Internationalization and Localization

- ***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.
- ***Crowdsourcing*** is a distributed problem-solving model where community members work on solutions to problems posted to the group.
- Crowdsourcing can be used to engage localization testers dispersed around the globe outside of the development environment.

Security Testing

- 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.
- On the client-side, vulnerabilities can often be traced to pre-existing 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.

Performance Testing

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point 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?

Load Testing

- The intent is to determine how the WebApp and its server-side environment will respond to various loading conditions.

  N, number of concurrent users
  T, number of on-line transactions per unit of time
  D, data load processed by server per transaction

- Overall throughput, *P*, is computed in the following manner:

  - $P = N \times T \times D$

Stress Testing

- ***Stress testing*** for mobile apps attempts to find errors that occur under extreme operating conditions such as:

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.
5. storing inordinately large quantities of data on the device.

Creating Weighted Device Platform Matrix

- To mirror real-world conditions, the demographic characteristics of testers should match those of targeted users, as well as those of their devices.
- A ***weighted device platform matrix*** (WDPM) helps ensure that test coverage includes each combination of mobile device and context variables.

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 (for example, 0 to 10) to indicate the relative importance of each operating system and each device.
4. compute the product of each pair of rankings and

Weighted Device Platform Matrix

- **TABLE 21.1 Weighted Device Platform Matrix**

		OS1	OS2	OS3
	Ranking	3	4	7
Device1	7	N/A	28	49
Device2	3	9	N/A	N/A
Device3	4	12	N/A	N/A
Device4	9	N/A	36	63

Testing AI Systems

- ***Static testing*** is a software verification technique that focuses on review rather than executable testing. It is important to ensure that human experts agree with the ways in which the developers have represented the information and its use in the AI system.
- ***Dynamic testing*** for AI systems is a validation technique that exercises the source code with test cases. The intent is to show that the AI system conforms to the behaviors specified by the human experts.
- ***Model-based testing*** (MBT) a black-box testing technique that uses the requirements model (user stories) as the basis for the generation of test cases from the behavioral model.

Testing Virtual Environments

- ***Acceptance tests*** are a series of specific tests conducted by the customer to uncover product errors before accepting the software from the developer.
- An ***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.
- A ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. The customer records all problems that are encountered and reports these at regular intervals.

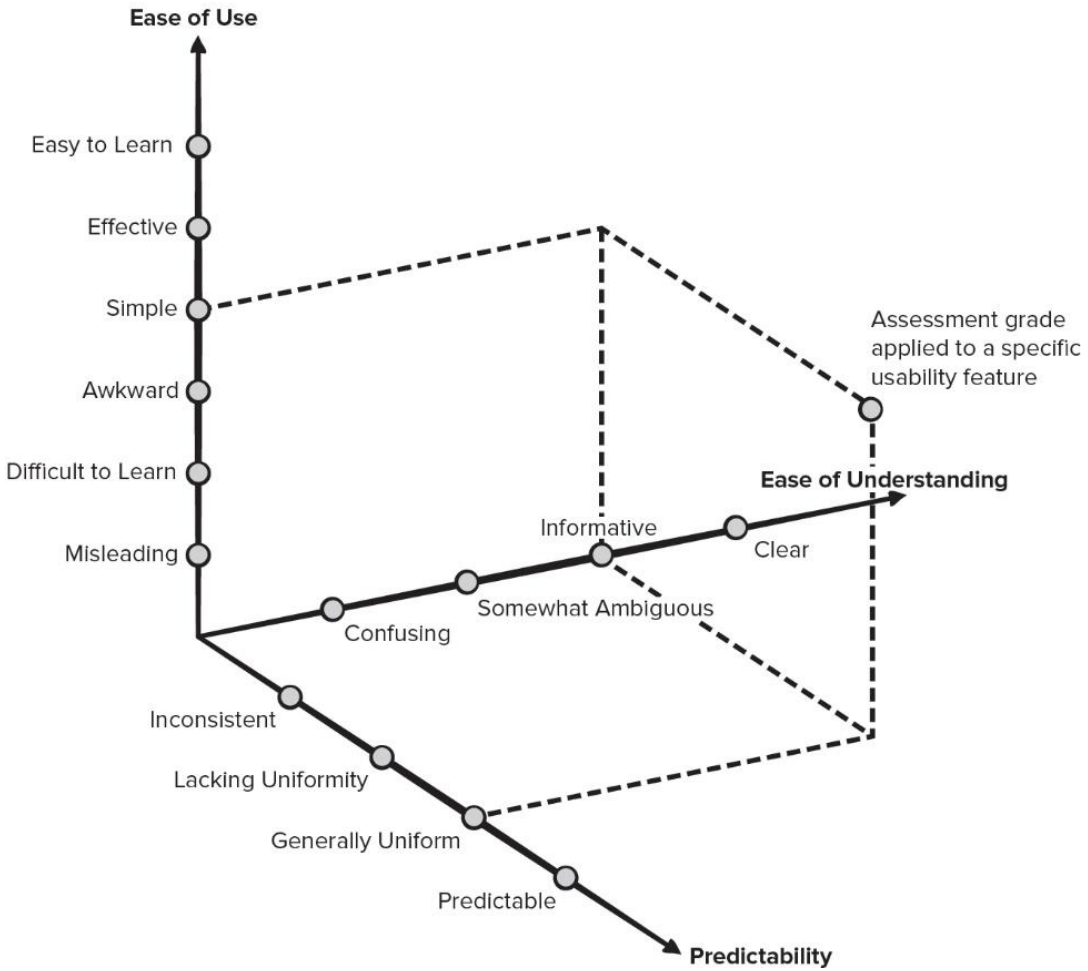
Usability Test Categories 1

- **Interactivity.** Are interaction mechanisms (for example, pull-down menus, buttons, widgets, inputs) easy to understand and use?
- **Layout.** Are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?
- **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 app?
- **Display characteristics.** Does the app make optimal use of screen size and resolution?

Usability Test Categories 2

- **Time sensitivity.** Can important features, functions, and content be used or acquired in a timely manner?
- **Feedback.** Do users receive meaningful feedback to their actions? Is the user's work interruptible and recoverable when a system message is displayed?
- **Personalization.** Does the app tailor itself to the specific needs of different user categories or individual users?
- **Help.** Is it easy for users to access help and other support options?
- **Accessibility.** Is the app accessible to people who have disabilities?
- **Trustworthiness.** Are users able to control how

Qualitative Usability Assessment



- Access the text alternative for slide images.

Accessibility Testing

- Ensure that non-text screen objects are also represented by a text-based description.
- Verify that color is not used exclusively to convey information to the user.
- Demonstrate that high contrast and magnification options are available for visually challenged users.
- Ensure that speech input alternatives have been implemented to accommodate users that may not be able to manipulate a keyboard, keypad, or mouse.
- Demonstrate that blinking, scrolling, or auto content updating is avoided to accommodate users with reading difficulties.

Playability Testing

- ***Playability*** is the degree to which a game or simulation is fun to play and usable by the user/player.
- Playability testing should be part of the usability testing for the virtual environment created by a MobileApp.
- Expert review should be supplemented by playability tests conducted by representative end users, as you might do for a beta or acceptance test.
- In a typical play test, the user might be given general instructions on using the app and the developers would then step back and observe players use of the game without interruption.

Documentation Testing

- Errors in help facilities or online program documentation can devastating to the acceptance of the program.
- Documentation testing should be an important part of every software test plan.

1. The first phase, technical review examines the document for editorial clarity.
2. The second phase, a live test, uses the documentation in conjunction with the actual program.