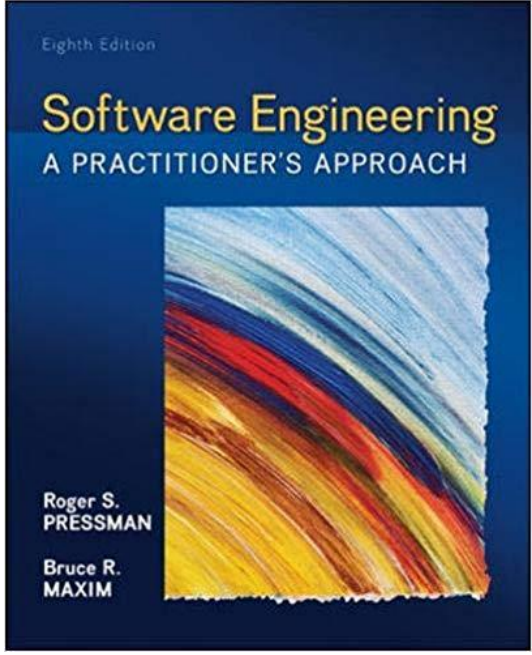


Software Testing

CS46 Software Engineering

Dr. Shilpa chaudhari

Department of Computer Science and Engineering
RIT, Bangalore

Outline of Unit-5

- Software Testing Strategies: A Strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Test Strategies for Object-Oriented Software, Test Strategies for WebApps, Test Strategies for MobileApp, Validation Testing, System Testing
  - Chapter - 22.1,22.2, 22.3,22.4, 22.5, 22.6, 22.7, 22.8
- Testing Conventional Applications: Software Testing Fundamentals, Internal and External Views of Testing, White-Box Testing, Basis Path Testing, Control Structure Testing, Black-Box Testing, Model-Based Testing, Testing Documentation and Help Facilities
  - Chapter - 23.1, 23.2, 23.3, 23.4, 23.5, 23.6, 23.7, 23.8
- Testing OOA and OOD Models, Object-Oriented Testing Strategies, Object-Oriented Testing Methods;
  - Chapter- 24.3, 24.4

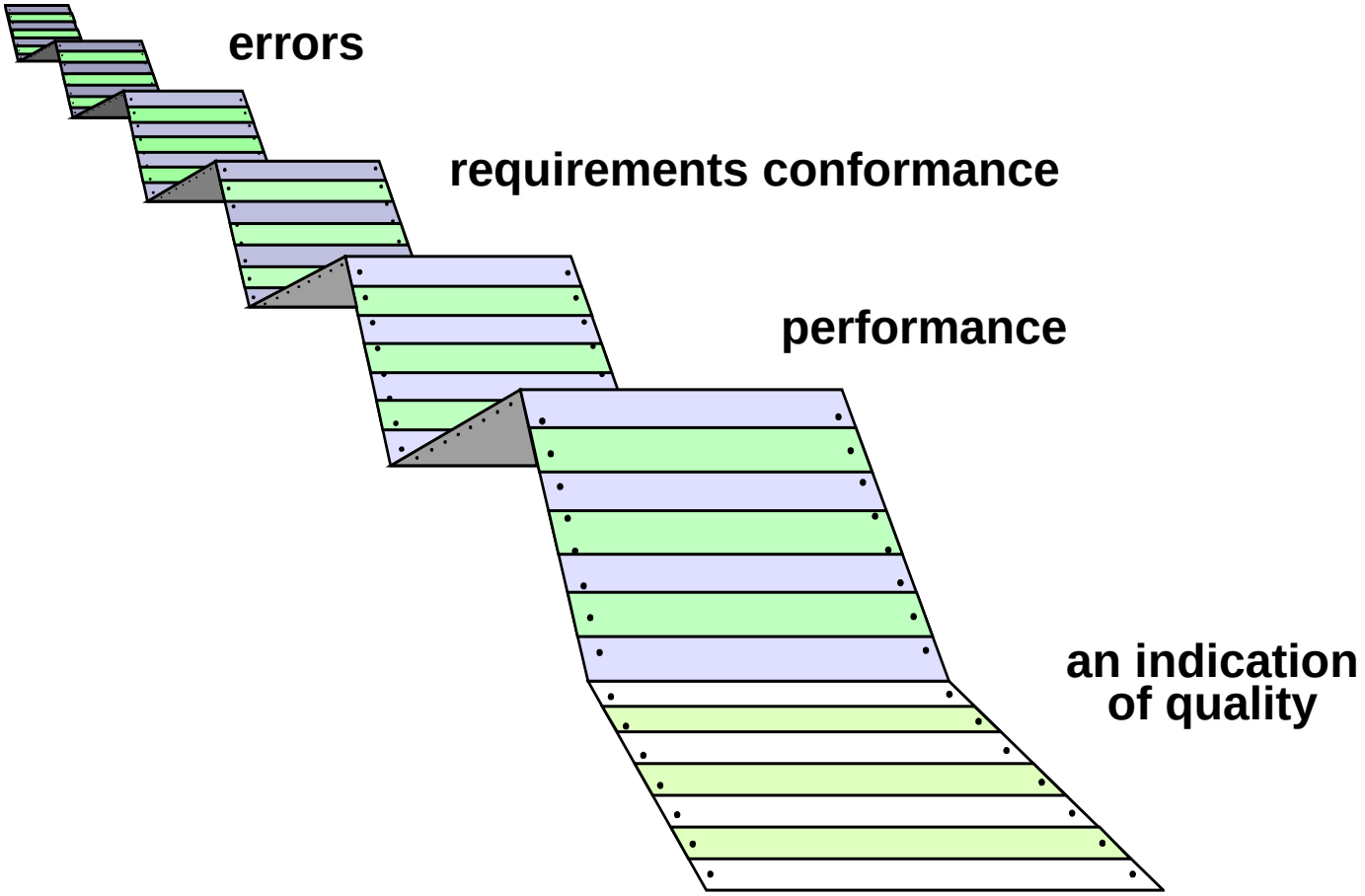
Software Testing Strategies

Software Testing

- **Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.**

4

What Testing Shows



5

Strategic Approach

- To perform effective testing, you should conduct effective technical reviews. 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.

6

Verification and Validation

- *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?"*

7

Who Tests the Software?

developer

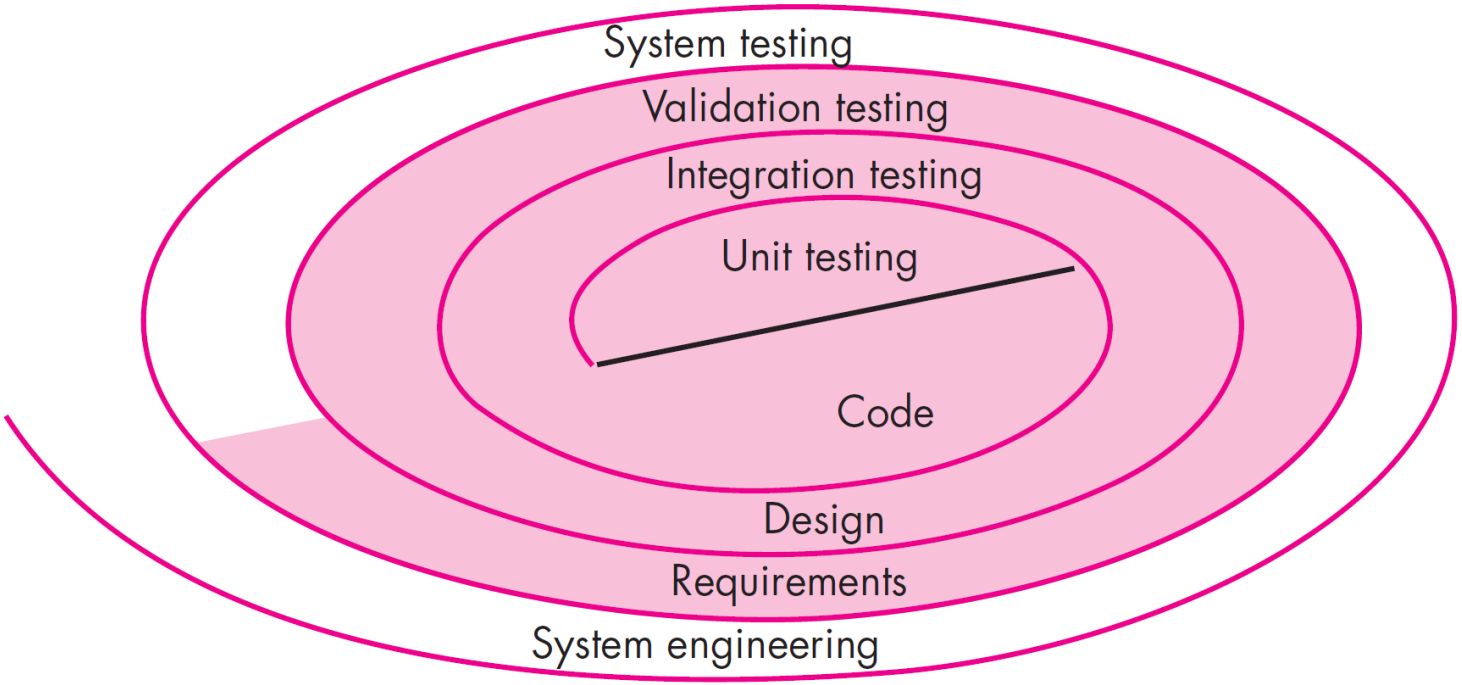
**Understands the system,
but will test "gently",
and is driven by "delivery"**

independent tester

**Must learn about the system,
but will attempt to break it,
and is driven by quality**

Testing Strategy

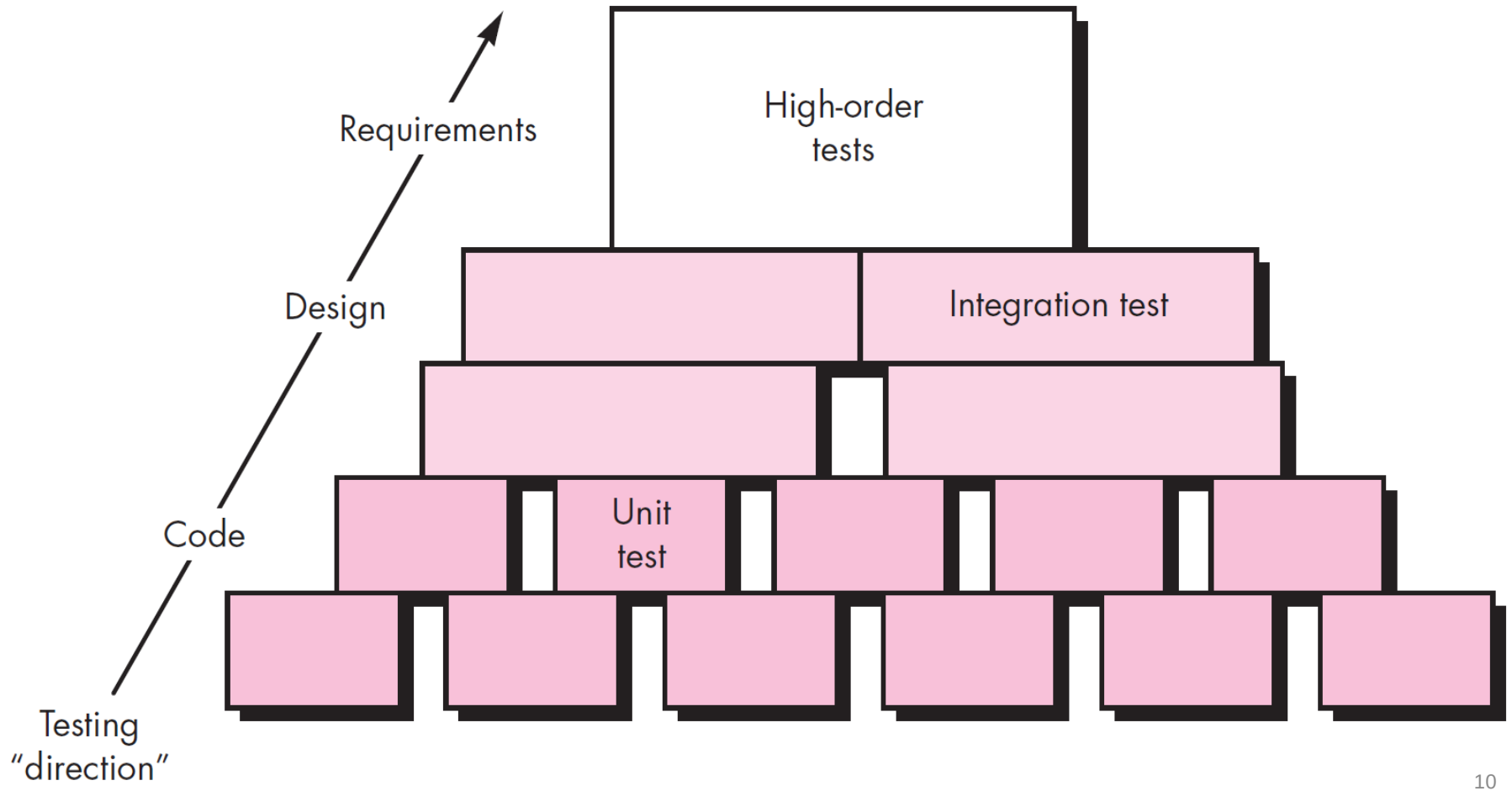
- System engineering -> requirements analysis -> design -> coding -> unit testing (component level) -> integration testing (architecture level) -> validation testing (conformance to requirements) -> system testing



9

Software Testing Steps

- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’



10

Strategic Issues

- A software testing strategy will succeed only when software testers:
  - Specify product requirements in a quantifiable manner long before testing commences.
  - State testing objectives explicitly.
  - Understand the users of the software and develop a profile for each user category.
  - Develop a testing plan that emphasizes “rapid cycle testing.”
  - Build “robust” software that is designed to test itself
  - Use effective technical reviews as a filter prior to testing
  - Conduct technical reviews to assess the test strategy and test cases themselves.
  - Develop a continuous improvement approach for the testing process.

11

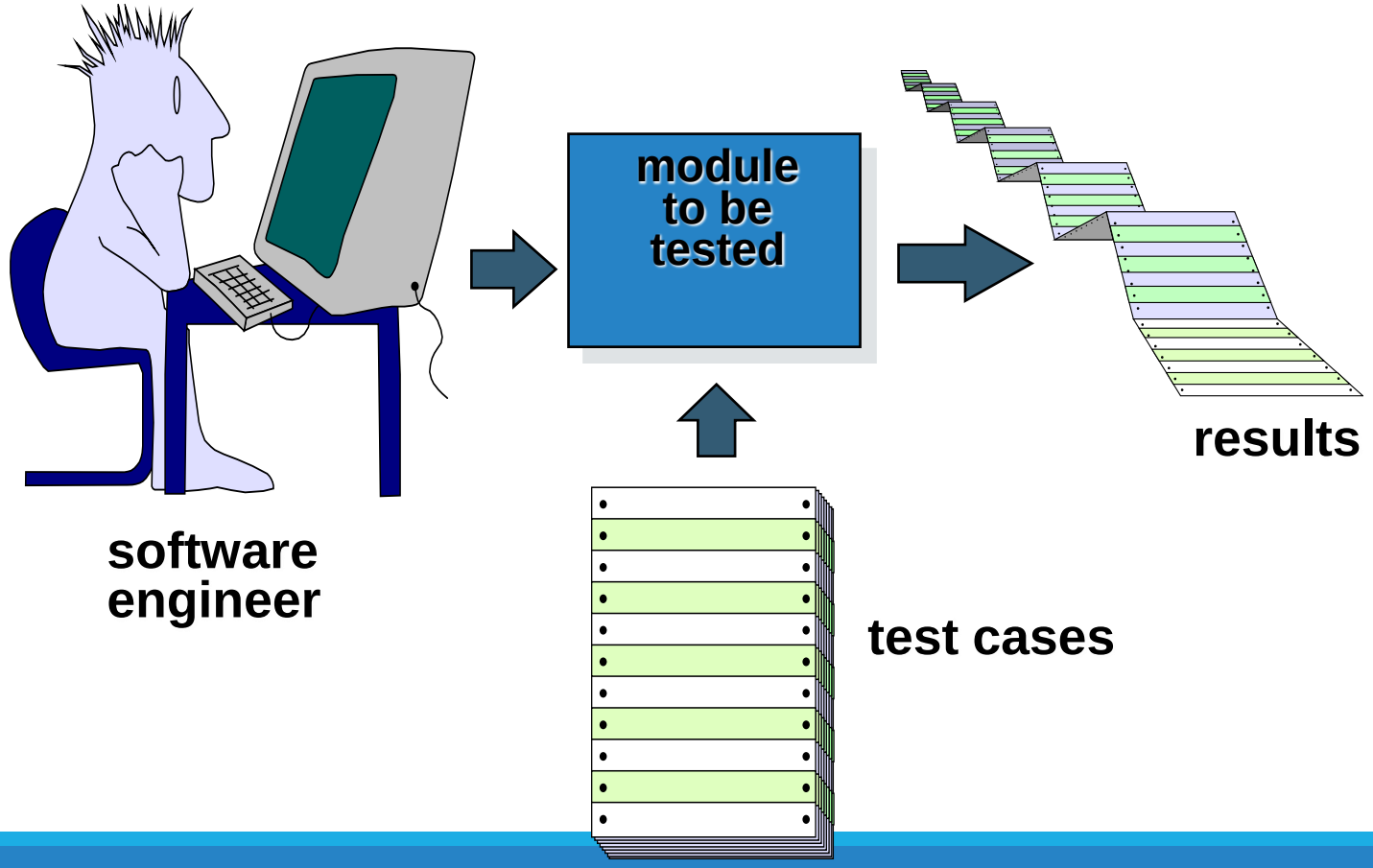
Test strategies for conventional software

- Unit testing
- Integration testing
- System testing

12

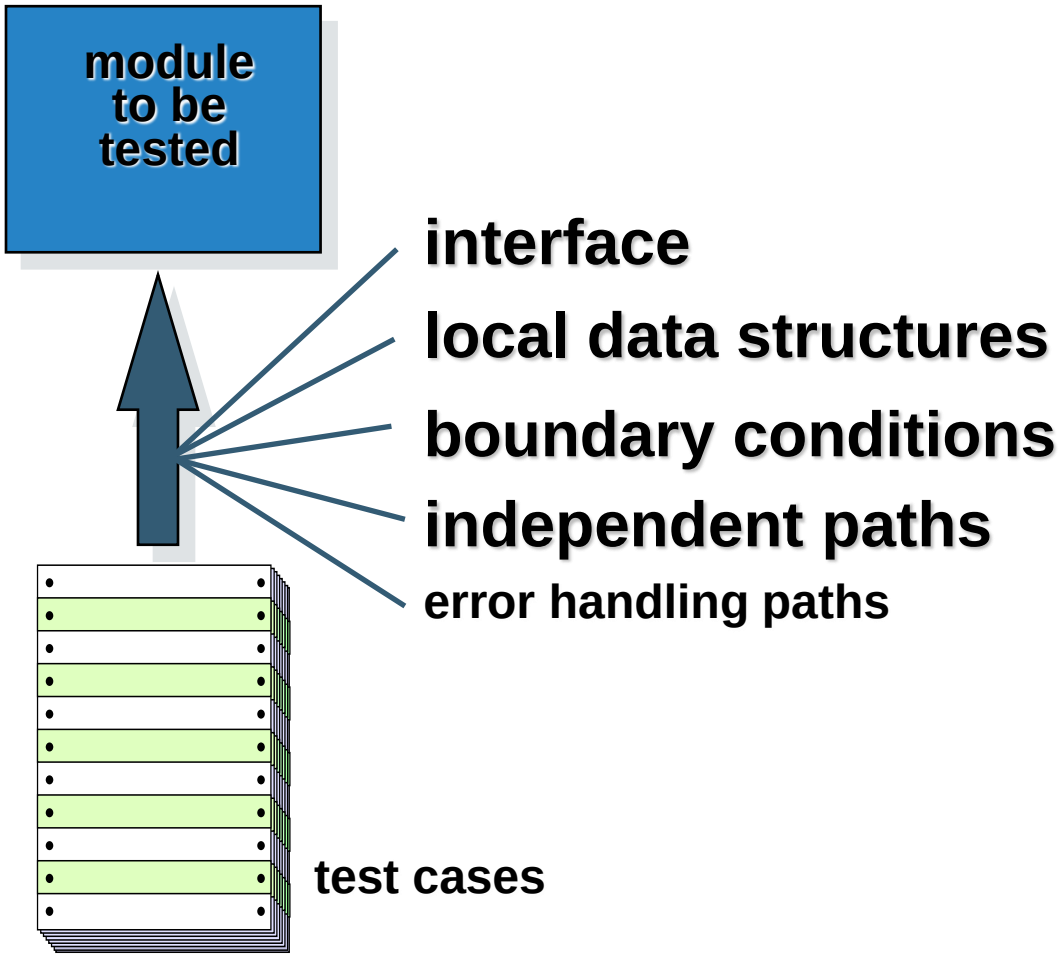
Unit Testing

- Focuses verification effort on the smallest unit of software design—the software component or module



13

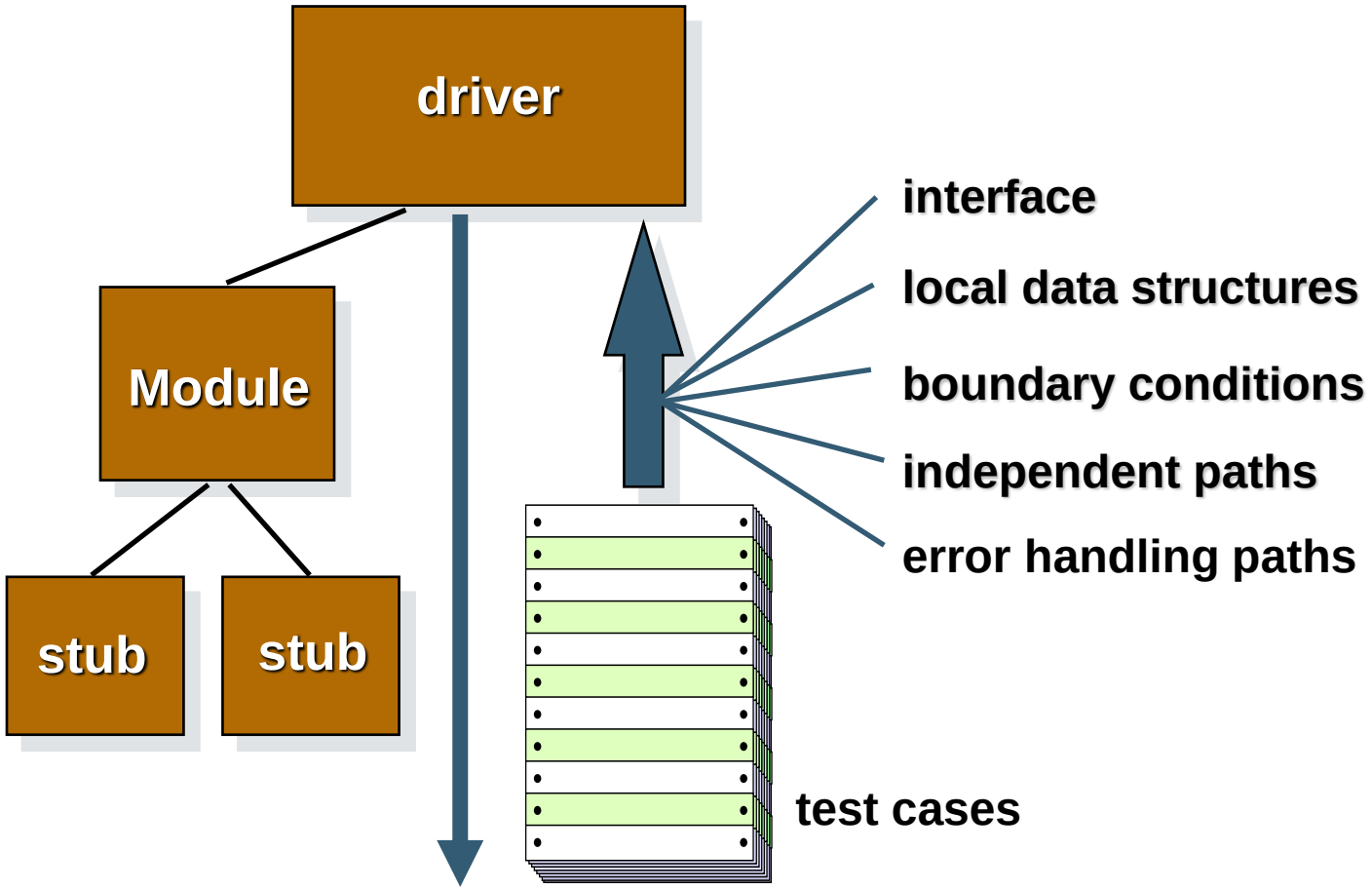
Unit Testing



14

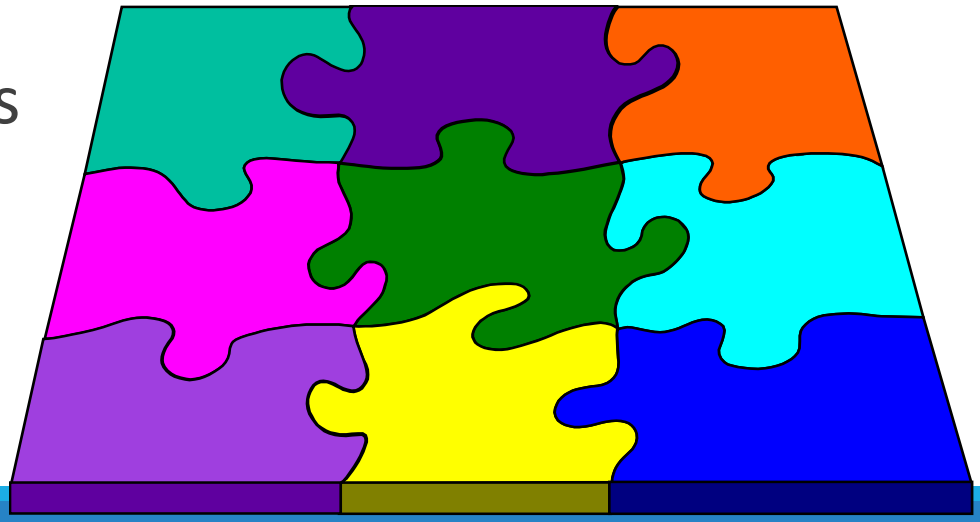
Unit Test Environment

- Drivers and stubs are testing "overhead" that must be coded but is not delivered with the final product.



Integration Testing

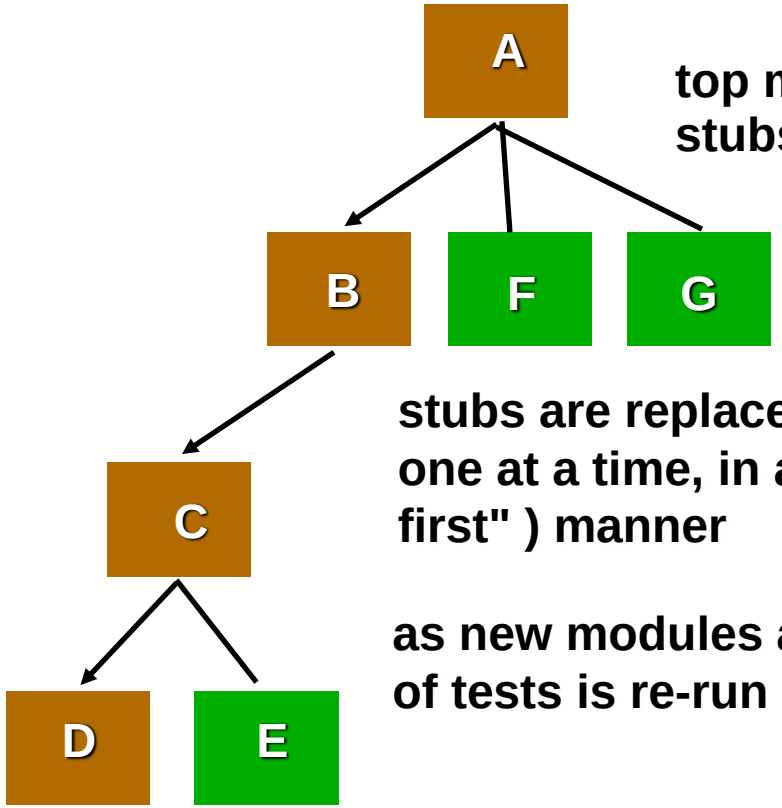
- A systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing.
- Takes unit-tested components and build a program structure.
- The “big bang” approach combine all components and then tests the entire program as a whole.
- In incremental integration approaches, the program is constructed and tested in small increments, where errors are easier to isolate and correct.



16

Top Down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).



top module is tested with stubs

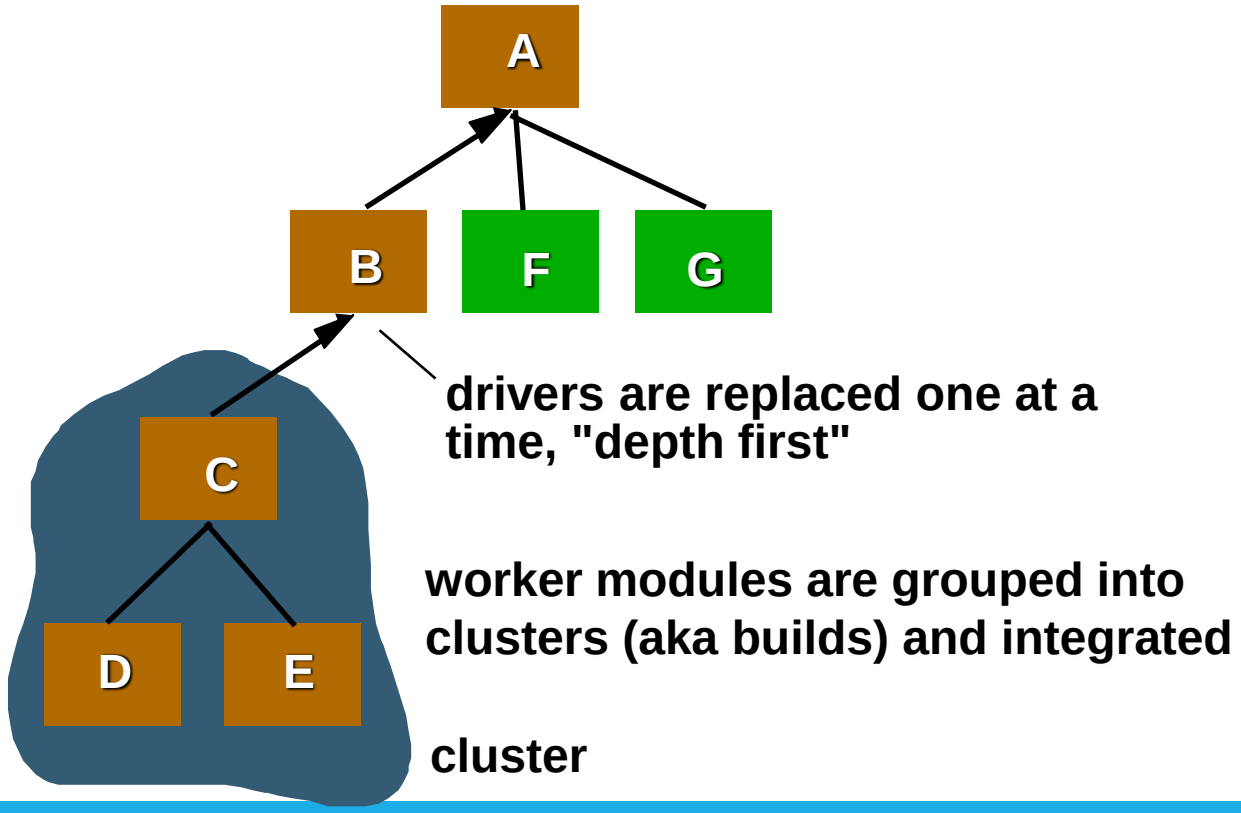
stubs are replaced with actual component one at a time, in a "depth first" (or "breadth first") manner

as new modules are integrated, some subset of tests is re-run (regression testing)

17

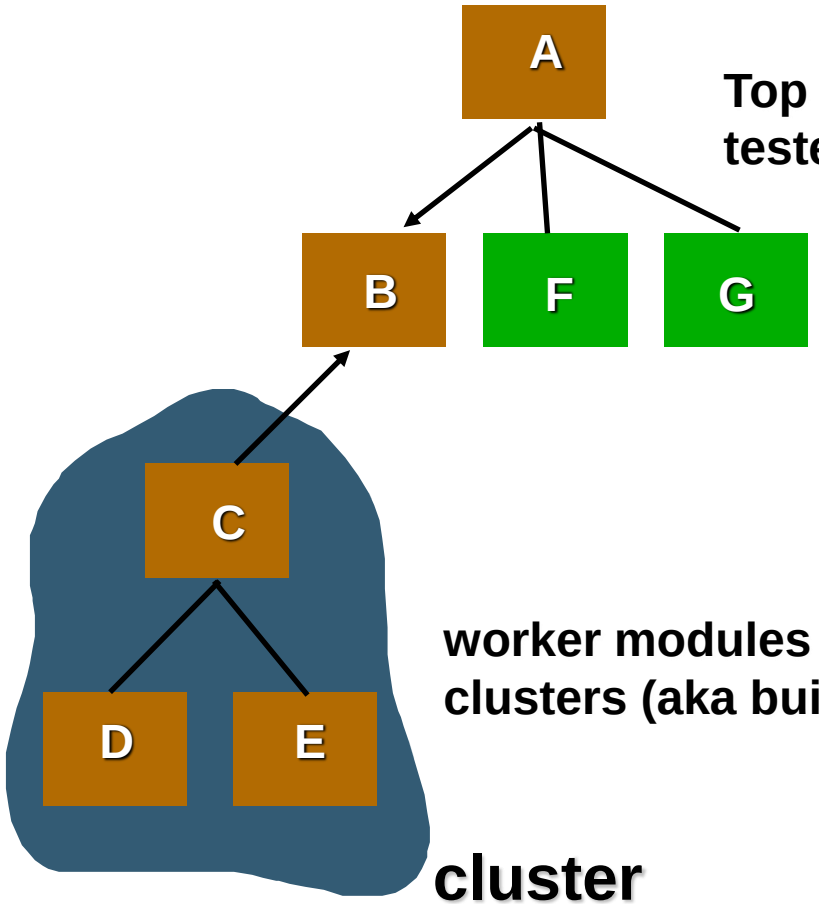
Bottom-Up Integration

- Begins construction and testing with *atomic modules* at the lowest levels in the program structure.
- A driver (a control program for testing) for each cluster is written to coordinate test-case input and output.



18

Sandwich Testing



Top modules are tested with stubs

worker modules are grouped into clusters (aka builds) and integrated

19

Comparison of integration approaches

Test Strategy	Method	Goal	Disadvantages
Top-Down	Incremental	Exercise critical code to improve reliability	Scaffolding takes time; Constant change may introduce new faults
Bottom-Up	Incremental	Perfect parts; If part work, whole should work	Functional flaws found late cause delays; Faults across modules difficult to trace and find

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.

21

Smoke Testing

- A common approach for creating "*daily builds*" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

22

General Testing Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify that specified performance bounds are tested

23

Object-Oriented Testing

- Begins by evaluating the correctness and consistency of the analysis and design models
- Testing strategy changes
  - the concept of the ‘unit’ broadens due to encapsulation (each class and each instance of a class package data and the operations that manipulate these data)
  - integration focuses on classes and their execution across a ‘thread’ (a set of classes that respond to an input or event) or in the context of a usage scenario
- Test case design draws conventional methods, but also encompasses special features (e.g., not advisable to test operations in isolation)

24

OO Testing Strategy

- Class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- Integration applied three different strategies
  - Thread-based testing—integrates the set of classes required to respond to one input or event for the system
  - Use-based testing—constructs the system by testing *independent* classes that use very few (if any) *server* classes, and then *dependent* classes in the next layer
  - Cluster testing—a cluster of collaborating classes is exercised to uncover errors in the collaborations

25

High Order Testing

- Validation testing—focuses on software requirements
- System testing—focuses on system integration
- Alpha/Beta testing—focuses on customer usage
- Recovery testing—forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing—verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing— executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing—tests the run-time performance of software within the context of an integrated system

26

WebApp Testing - I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.

27

WebApp Testing - II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- 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.

28

MobileApp Testing

- User experience testing – ensuring app meets stakeholder usability and accessibility expectations
- Device compatibility testing – testing on multiple devices
- Performance testing – testing non-functional requirements
- Connectivity testing – testing ability of app to connect any needed networks or Web services reliably
- Security testing – ensuring app meets stakeholder security expectations
- Testing-in-the-wild – testing app on user devices in actual user environments
- Certification testing – ensuring app meets the standards established by the app stores that will distribute it

29

Testing Conventional Applications

Software testing fundamentals

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be
  - automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

Test Characteristics

What is 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

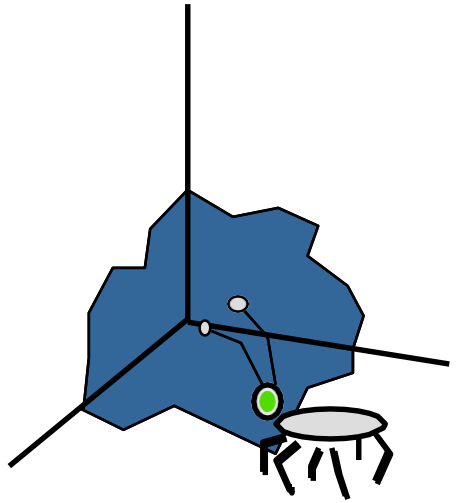
Internal and External Views

- Any engineered product (and most other things) can be tested in one of two ways:
  - 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;
  - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



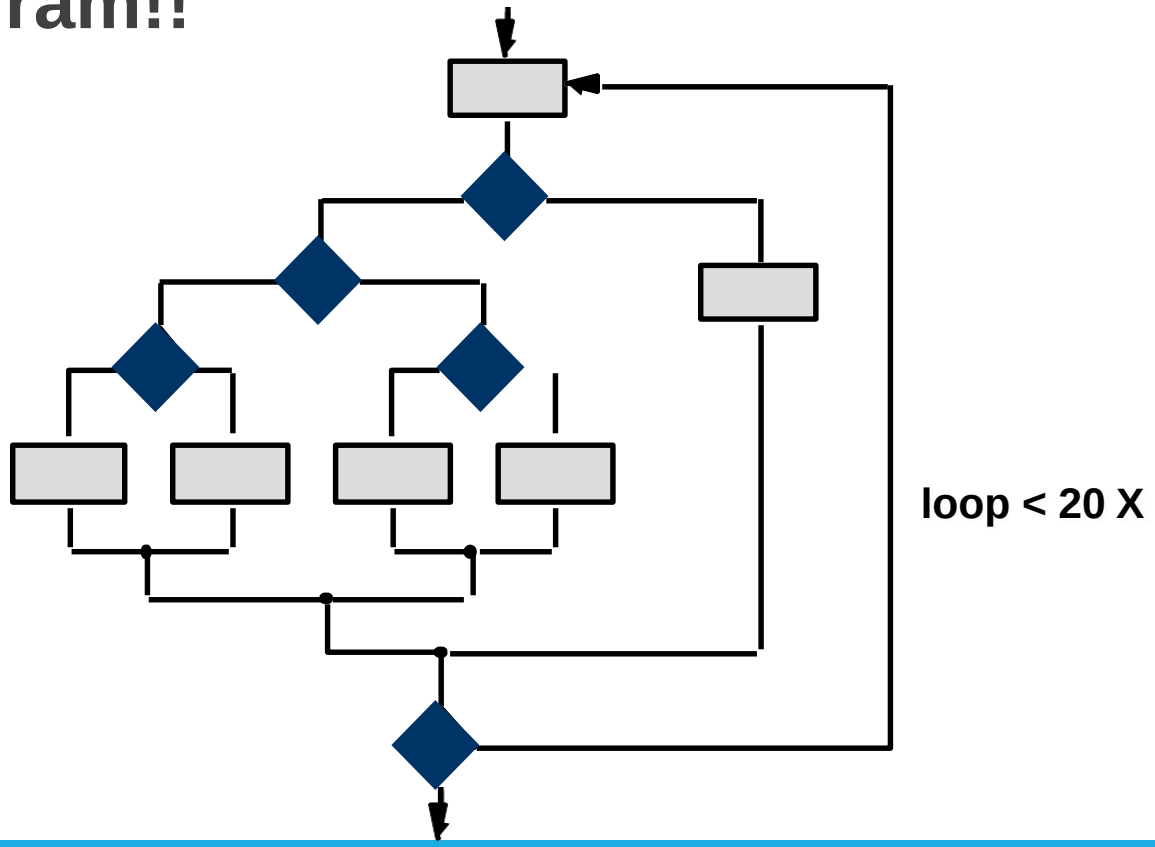
OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

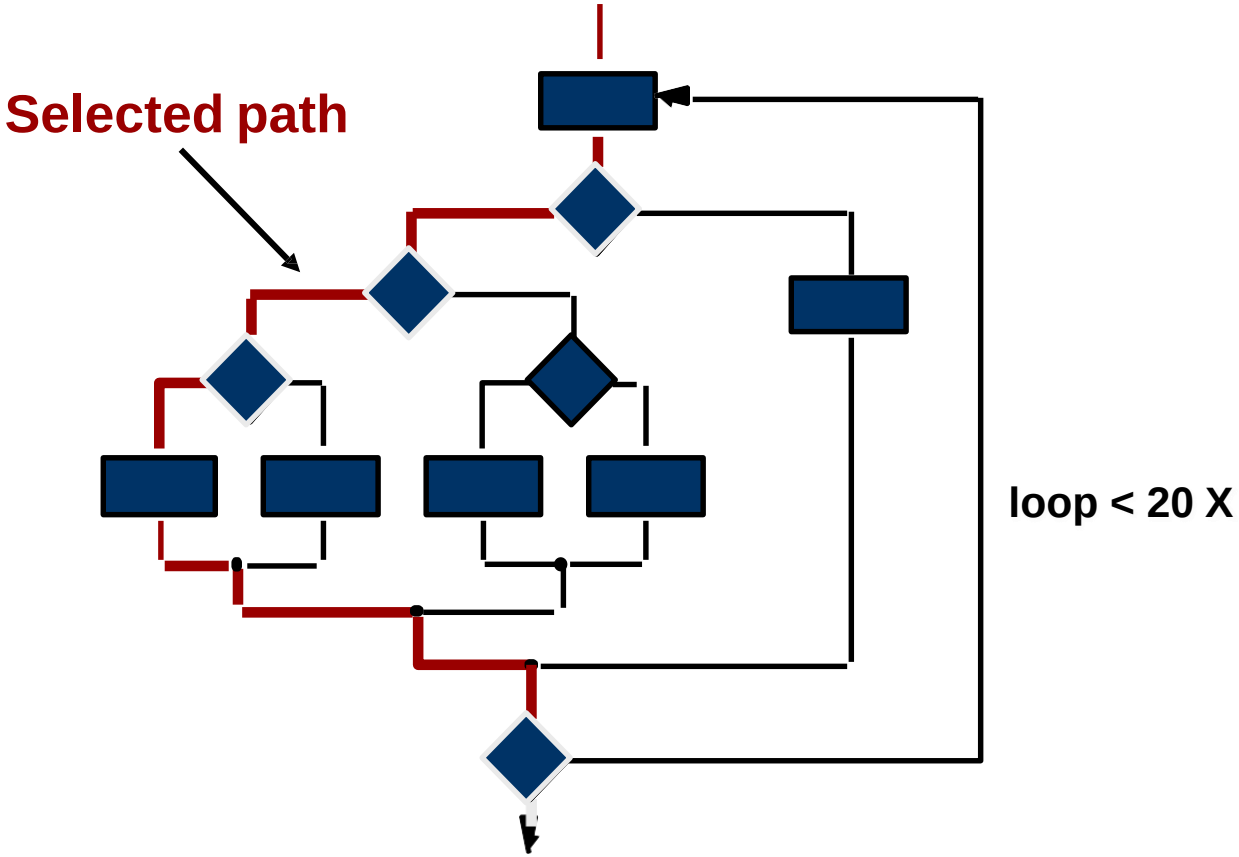
CONSTRAINT **with a minimum of effort and time**

Exhaustive Testing

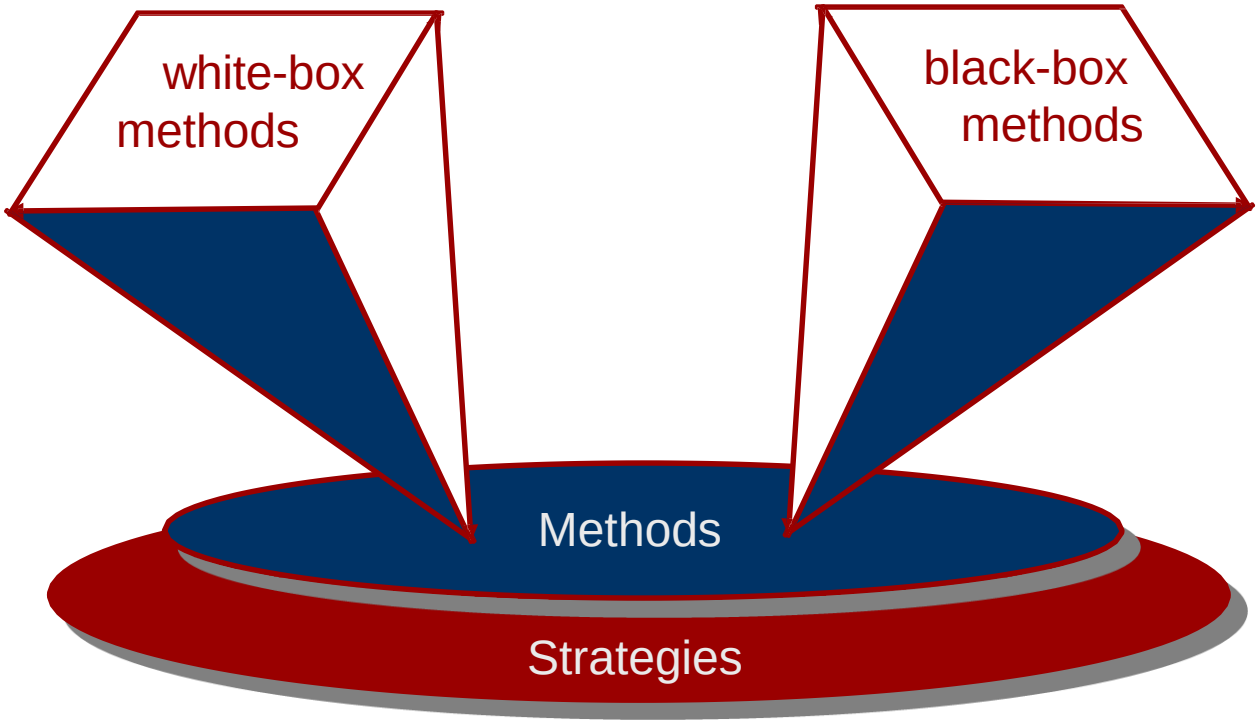
- **There are 10 possible paths! If we execute one test per millisecond, it would take many years to test this program!!**



Selective Testing



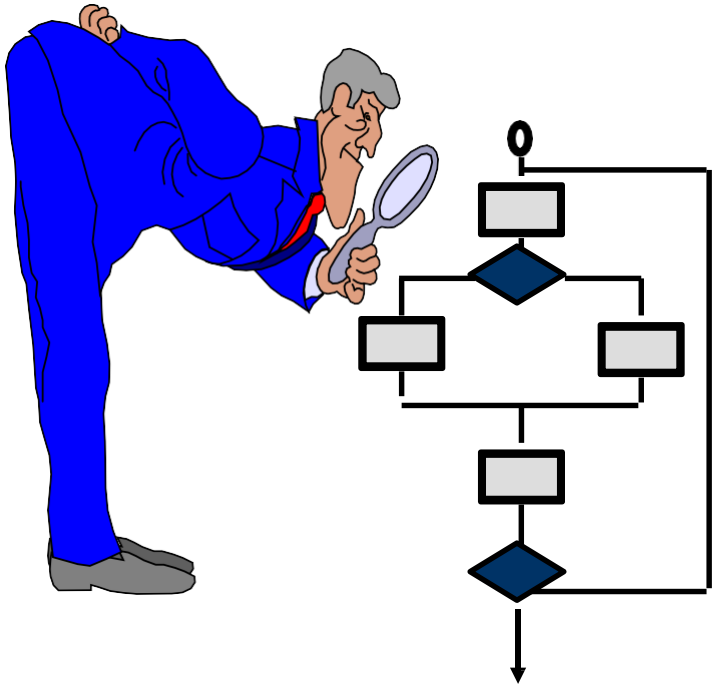
Software Testing



White-Box Testing

- **... our goal is to ensure that all statements and conditions have been executed at least once ...**

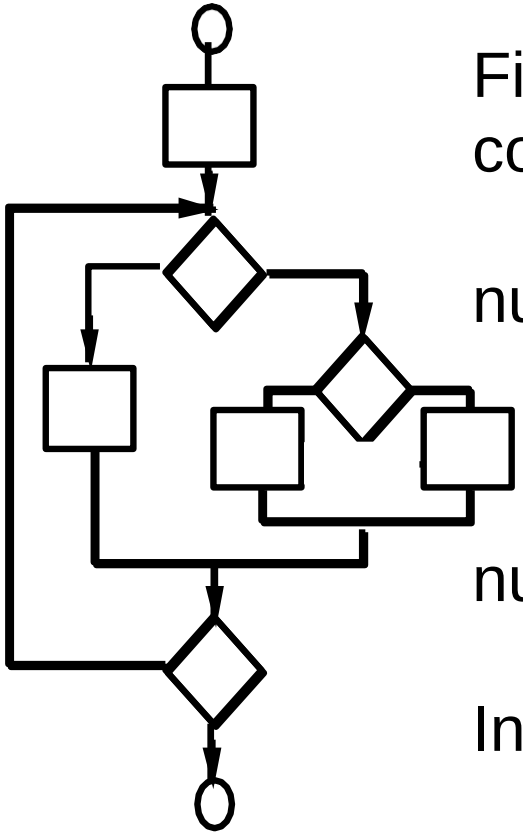
- Statement Coverage
- Branch Coverage
- Condition Coverage
- Multiple Condition Coverage
- Path Coverage



Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

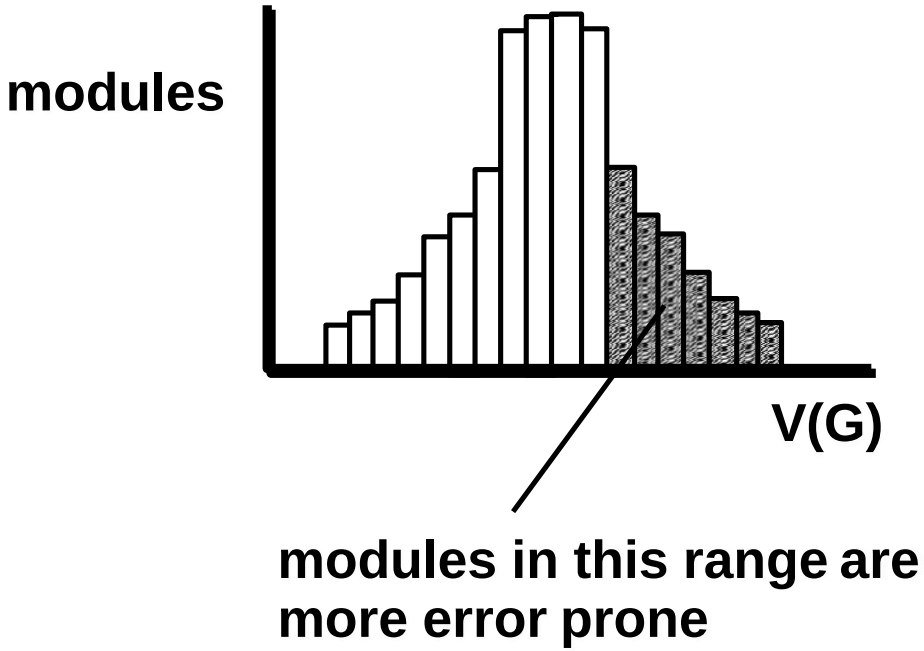
number of enclosed areas + 1

In this case, V(G) = 4

40

Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



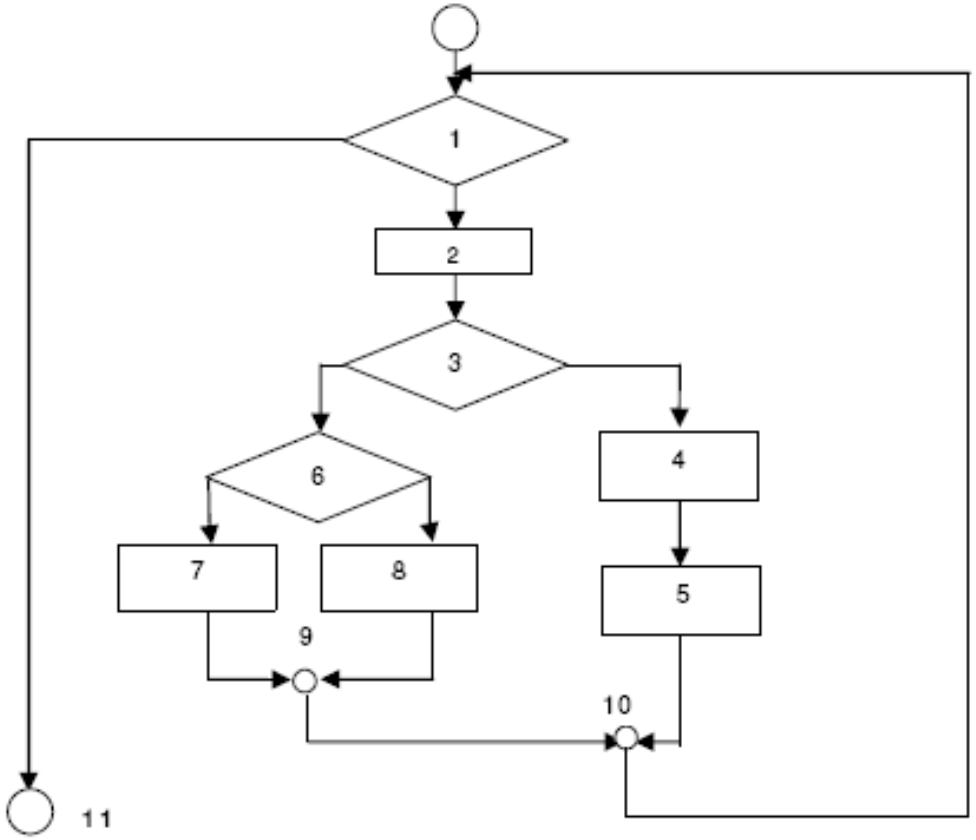
41

Cyclomatic Complexity

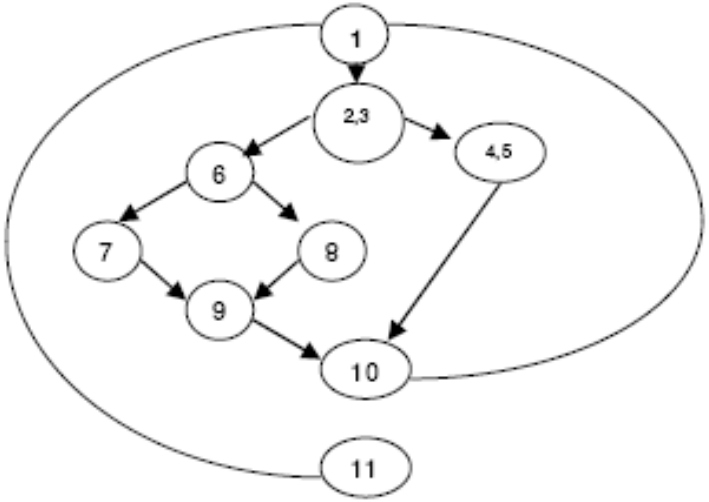
- Software metric that provides a quantitative measure of the logical complexity of the program
- The value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us an upper bound for the number of test cases that must be identified to ensure that all statements have been executed at least once
  - an independent path is any path through the program that introduces at least one new set of processing statements or a new condition
- The Cyclomatic complexity is determined as
  - $V(G) = E - N + 2$
    - Where G is the flow graph with E number of edges and N number of nodes

Cyclomatic Complexity

Consider the flowchart below:

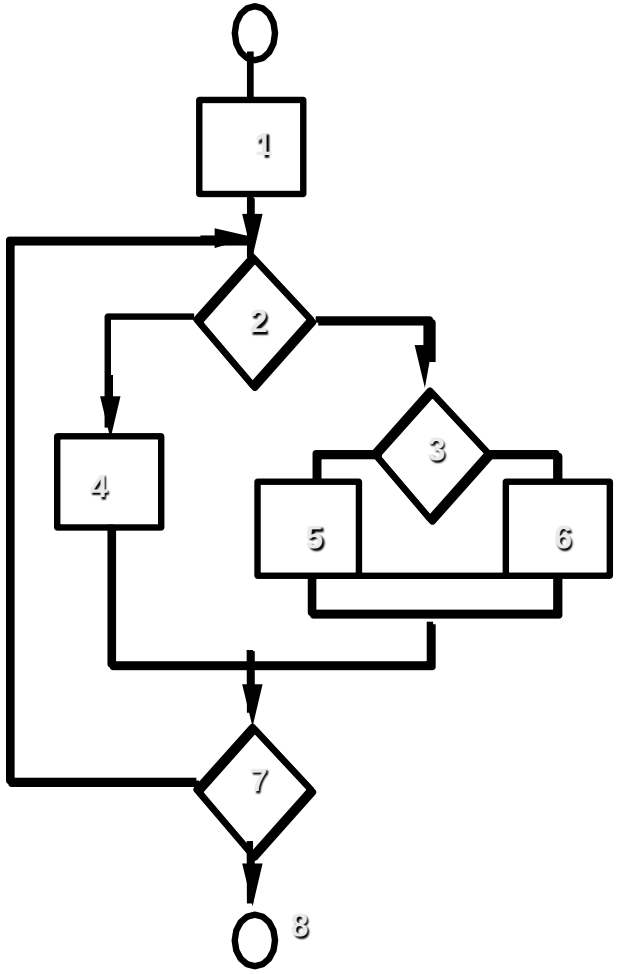


Flow graph may be drawn as follows:



number of edges E = 11
Number of nodes N = 9
V(G) = 11-9+2 = 4

Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

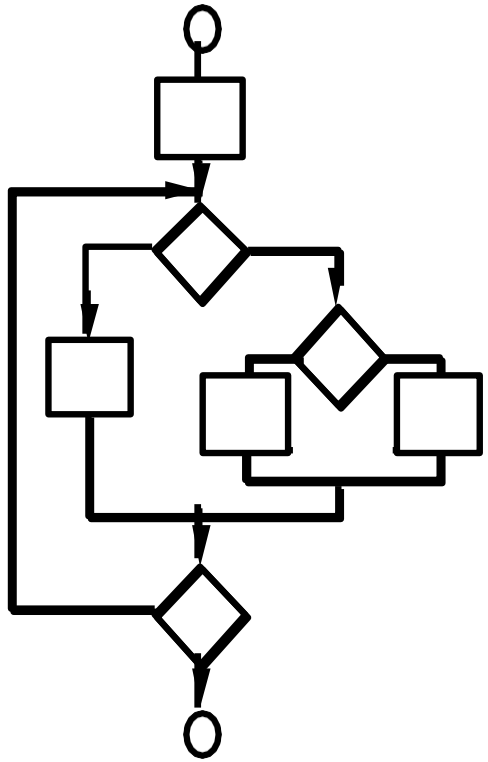
Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

44

Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

45

Deriving Test Cases

- *Summarizing:*
  - 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.

46

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

47

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

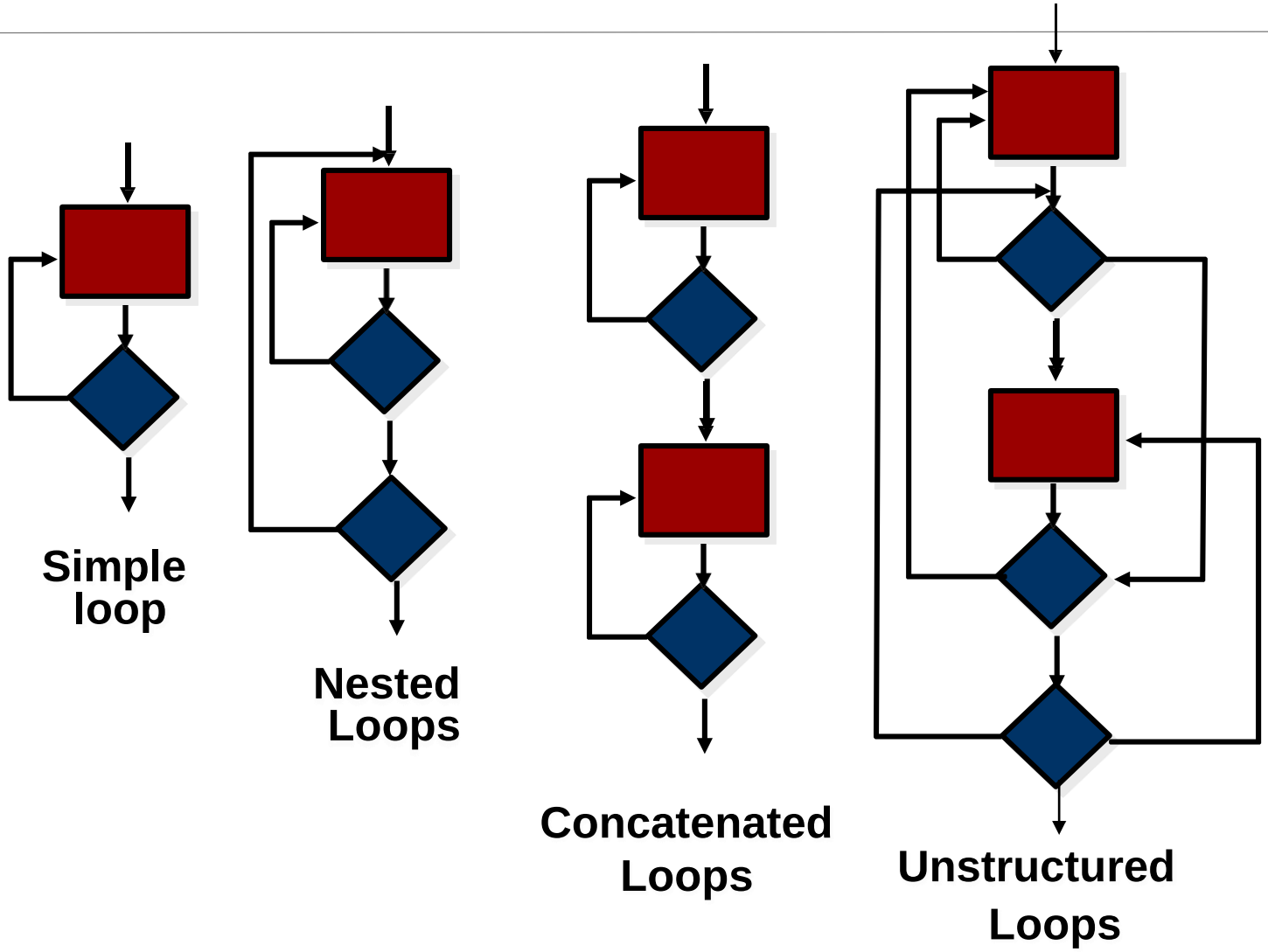
48

Data Flow Testing

- The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number
    - DEF(*S*) = {*X* | statement *S* contains a definition of *X*}
    - USE(*S*) = {*X* | statement *S* contains a use of *X*}
  - A *definition-use (DU) chain* of variable X is of the form [*X, S, S′*], where *S* and *S′* are statement numbers, *X* is in DEF(*S*) and USE(*S′*), and the definition of *X* in statement *S* is live at statement *S′*

49

Loop Testing



Loop Testing: Simple Loops

- Minimum conditions—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 m < n
  5. (n-1), n, and (n+1) passes through the loop
     - where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

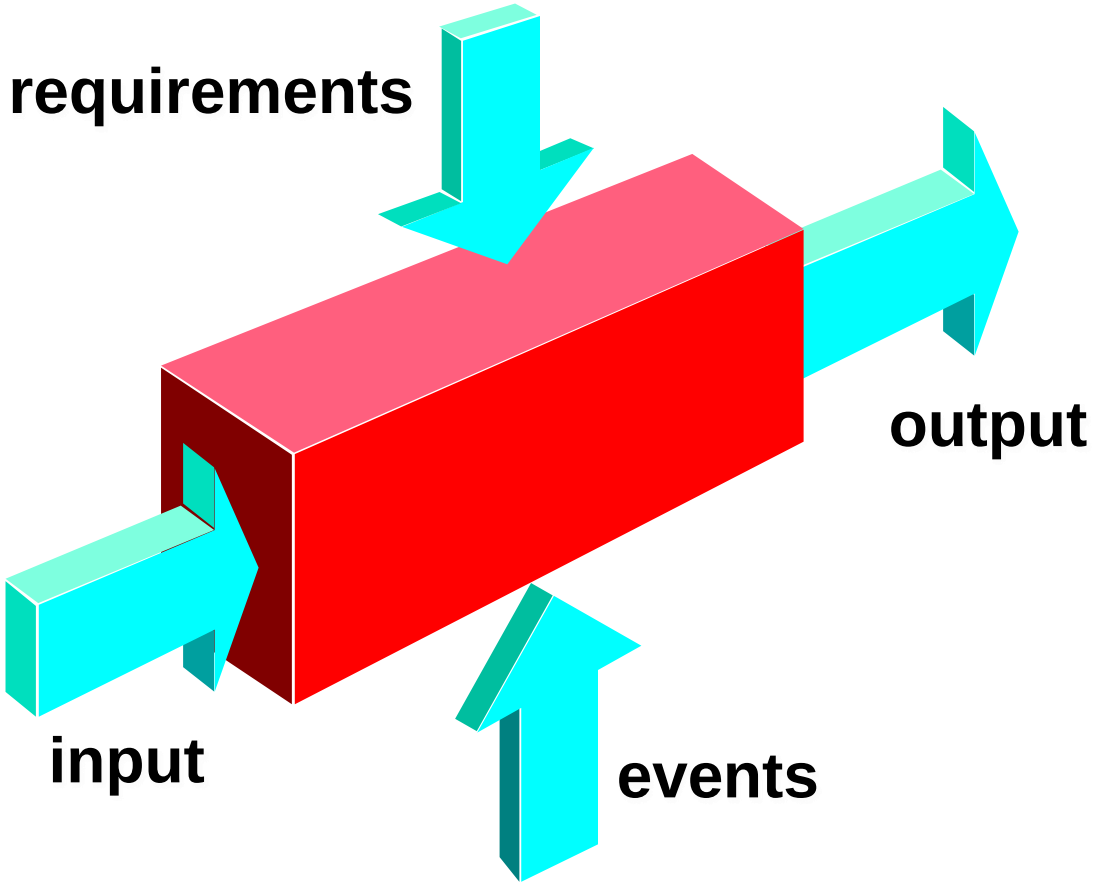
Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

52

Black-Box Testing

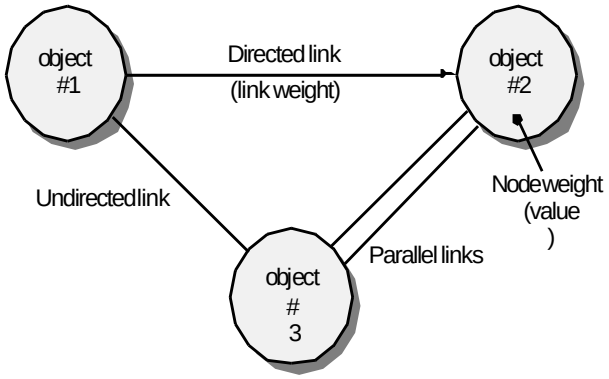


Black-Box Testing

- How is functional validity tested?
- How is 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?

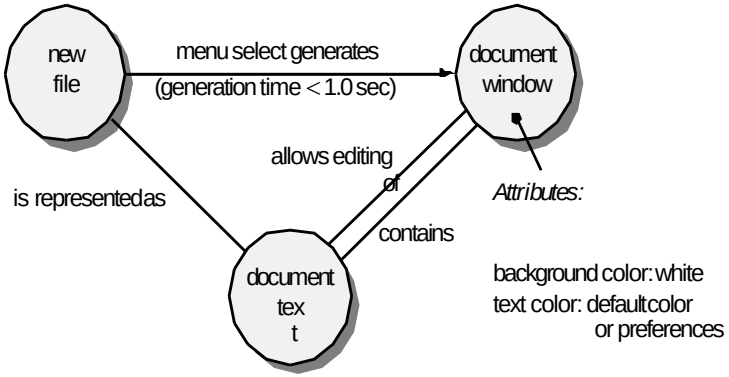
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects



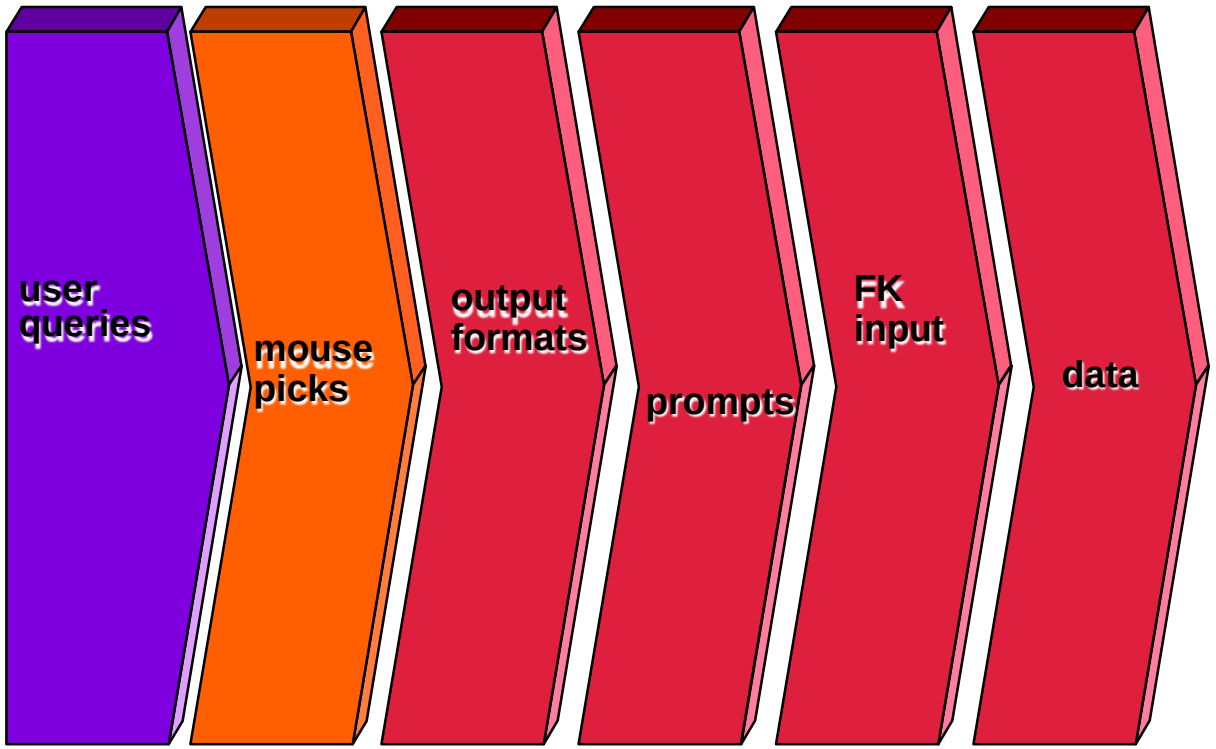
(a)

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

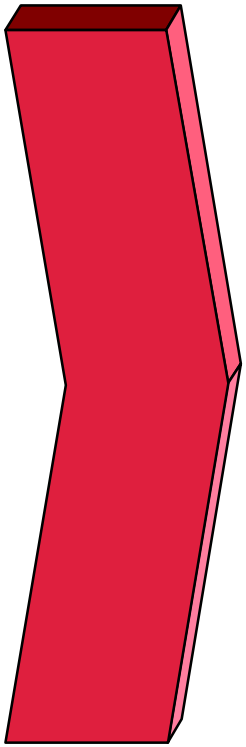


(b)

Equivalence Partitioning



Sample Equivalence Classes



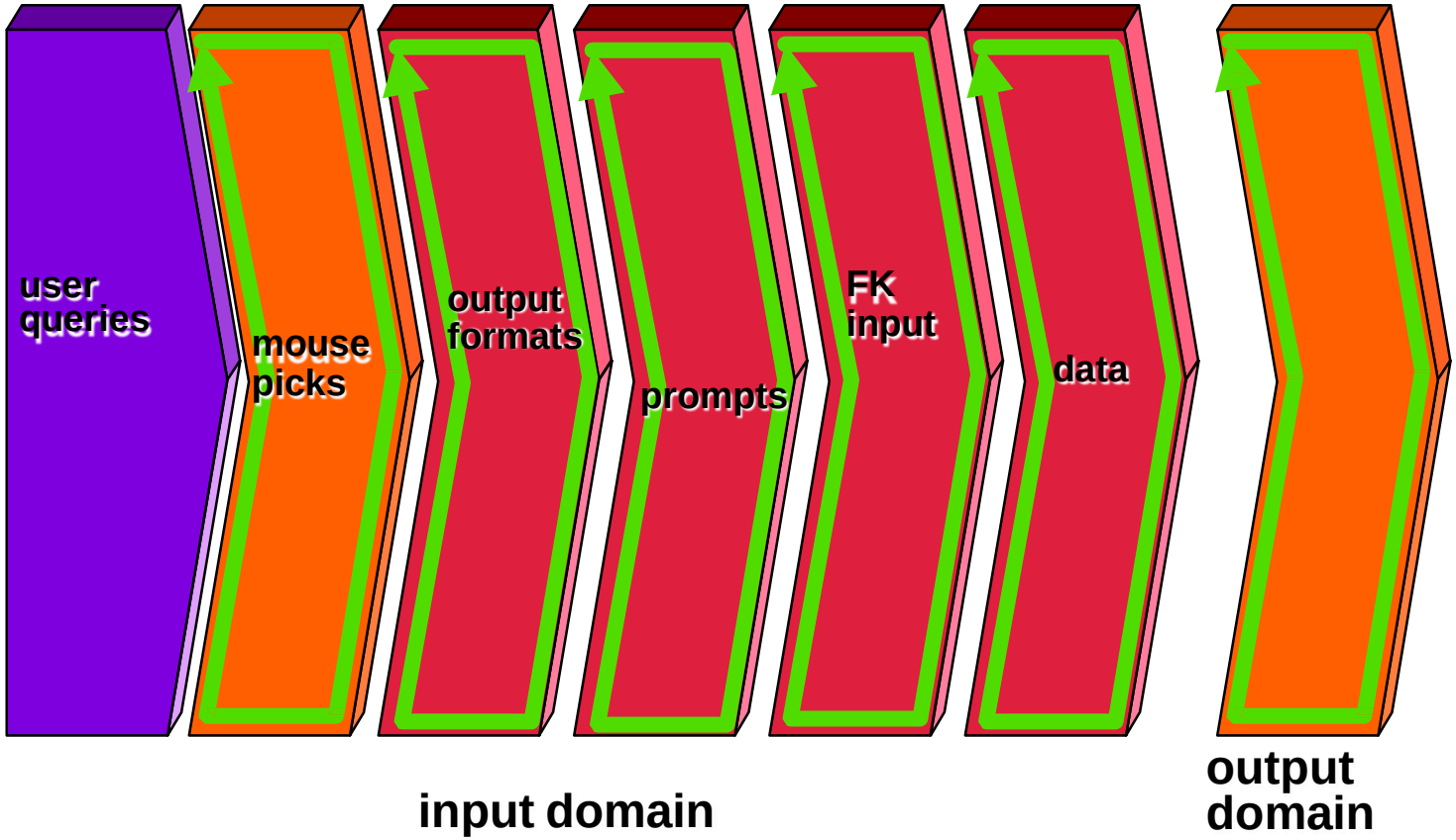
Valid data

- user supplied commands
- responses to system prompts file names
- computational data physical parameters bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

Invalid data

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

Boundary Value Analysis

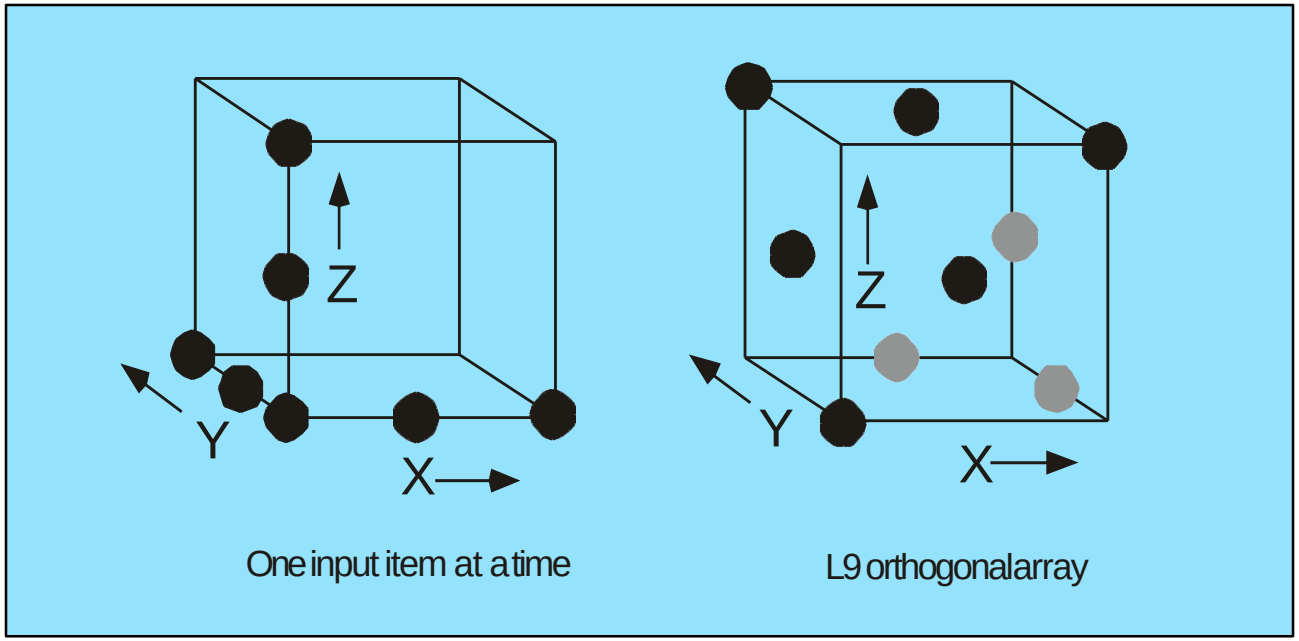


Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human- rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

Testing OOA and OOD Models

OO Testing Strategies

- **Unit testing**
  - the concept of the unit changes
  - the smallest testable unit is the encapsulated class
  - a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class
- **Integration Testing**
  - Thread-based testing integrates the set of classes required to respond to one input or event for the system
  - Use-based testing begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes.
    - After the independent classes are tested, the next layer of classes, called dependent classes
  - Cluster testing defines a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

OO Testing Strategies

- **Validation Testing**
- details of class connections disappear
- draw upon use cases that are part of the requirements model
- Conventional black-box testing methods can be used to drive validation tests

OOT Methods

- **Berard proposes the following approach:**
  1. Each test case should be uniquely identified and should be explicitly associated with the class to be tested,
  2. The purpose of the test should be stated,
  3. A list of testing steps should be developed for each test and should contain:
     a. a list of specified states for the object that is to be tested
     b. a list of messages and operations that will be exercised as a consequence of the test
     c. a list of exceptions that may occur as the object is tested
     d. a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
     e. supplementary information that will aid in understanding or implementing the test.

Testing Methods

- **Fault-based testing**
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects).
  - To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes.
  - In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - 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, then applying them and their variants as tests.