Software Testing Strategies

Chapter 17

Roger Pressman – 7th Edition

Software Testing

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

V & V

- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

Who Tests the Software?

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

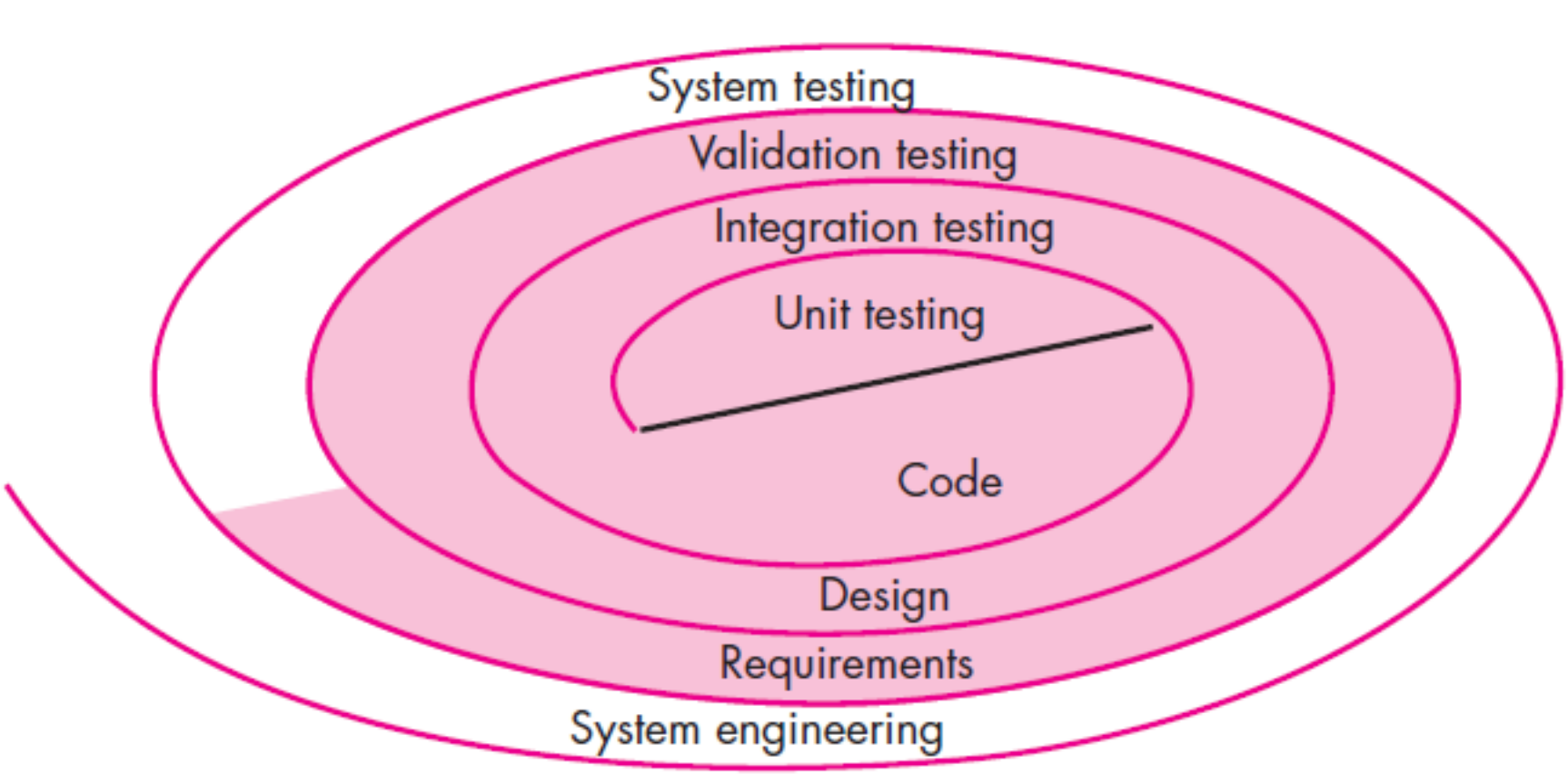
independent tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

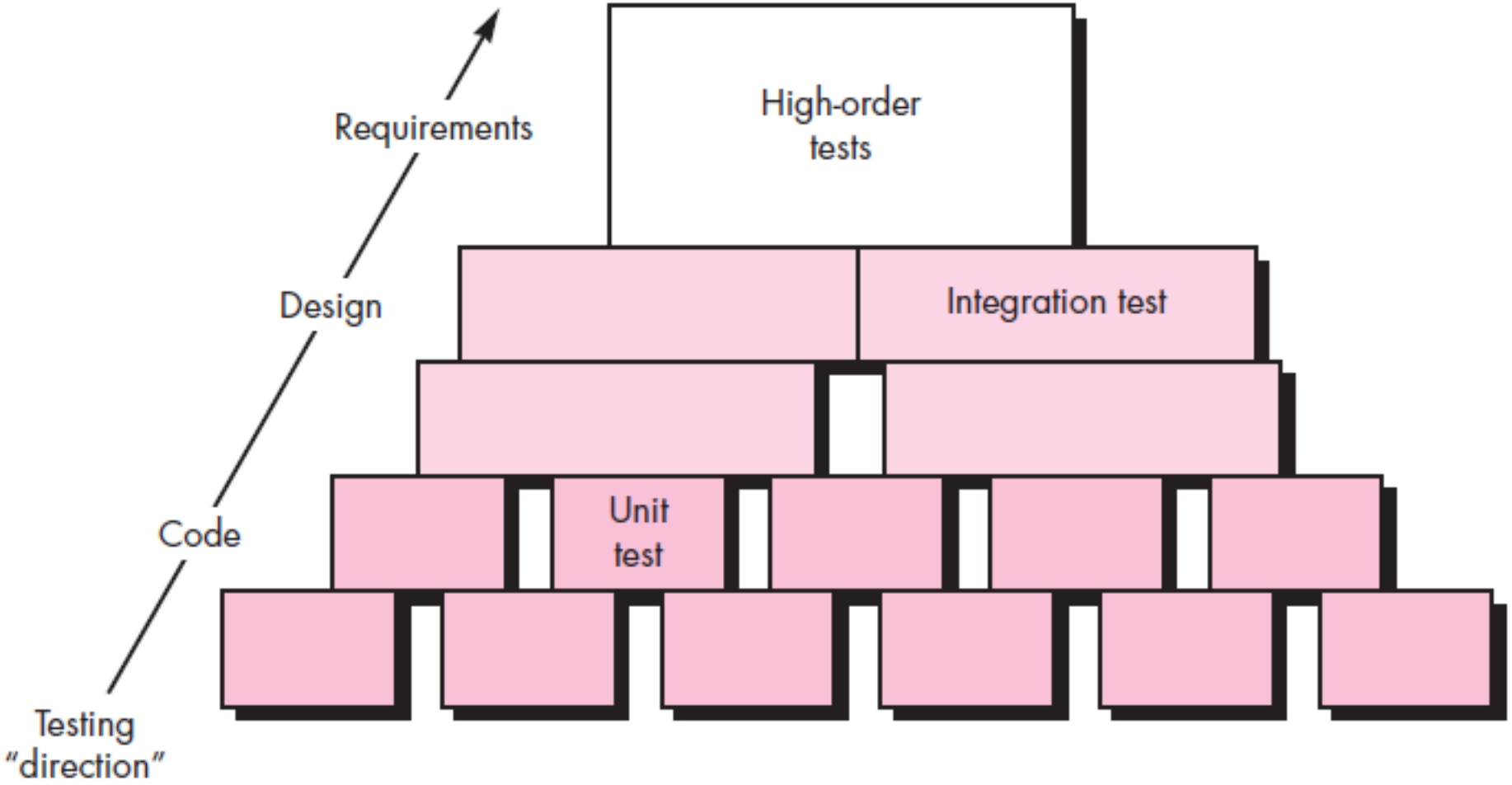
Testing Strategy



Development strategy vs. testing strategy

- Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.
- Moving inward along the spiral, you come to design and finally to coding.
- To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.
- To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

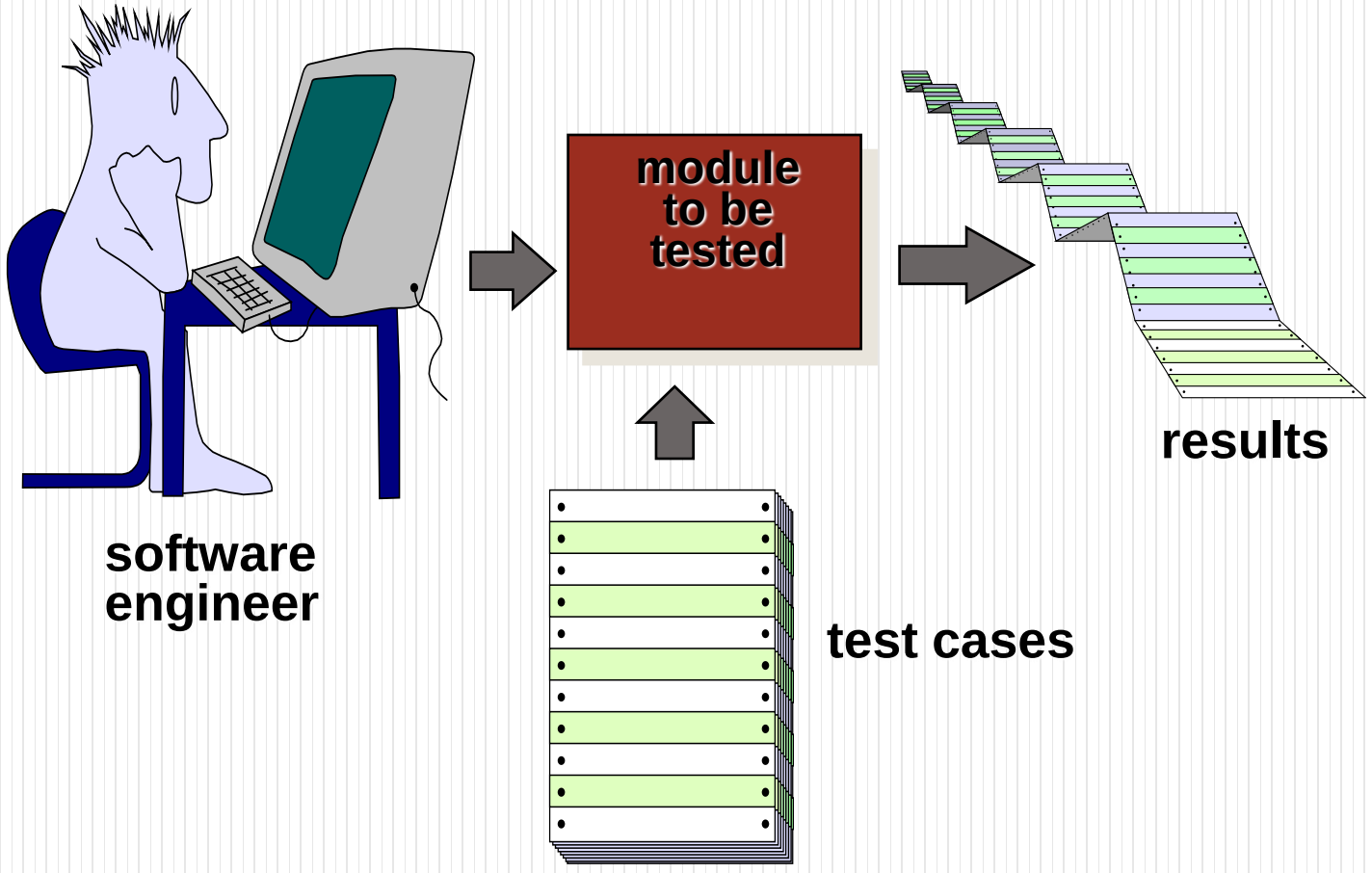
Software Testing steps



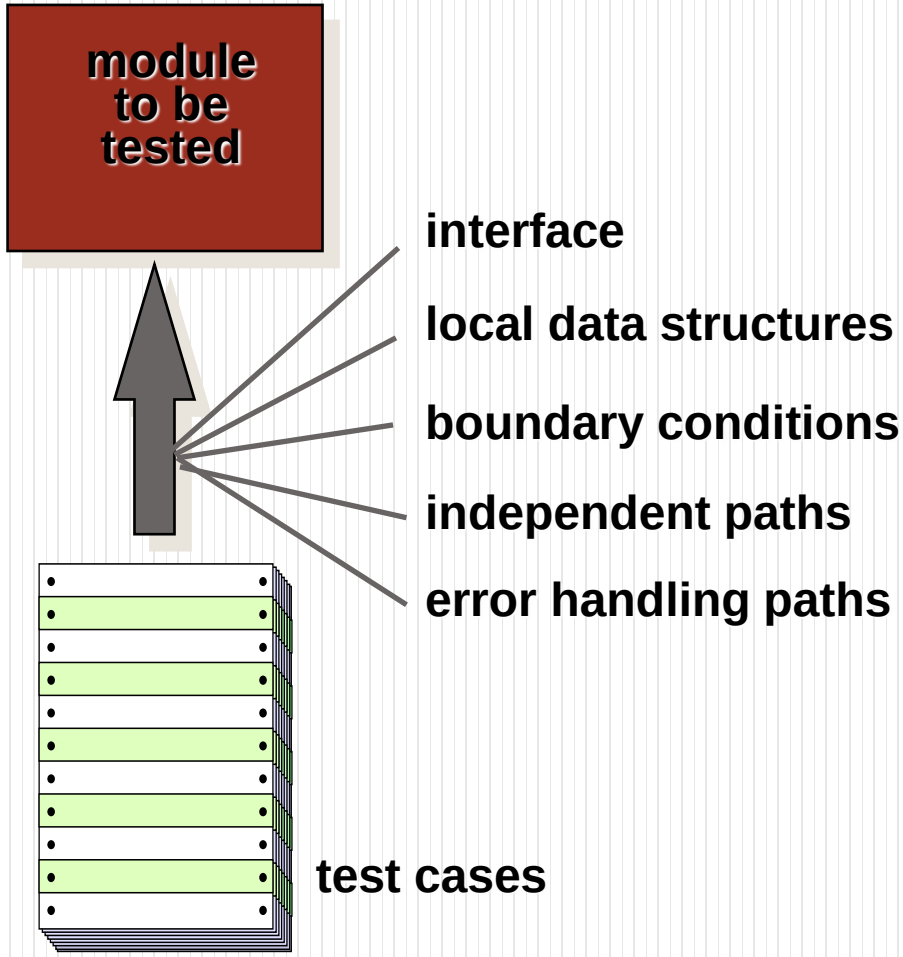
Testing Strategy

- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - Our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

Unit Testing

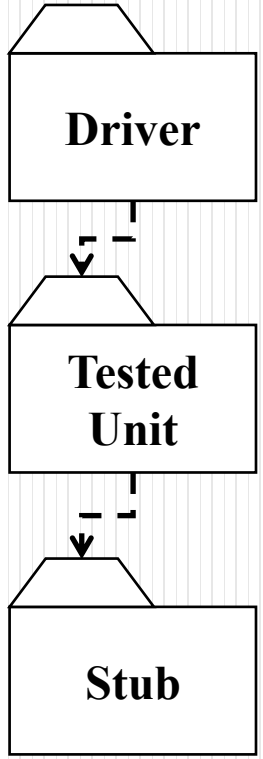


Unit Testing

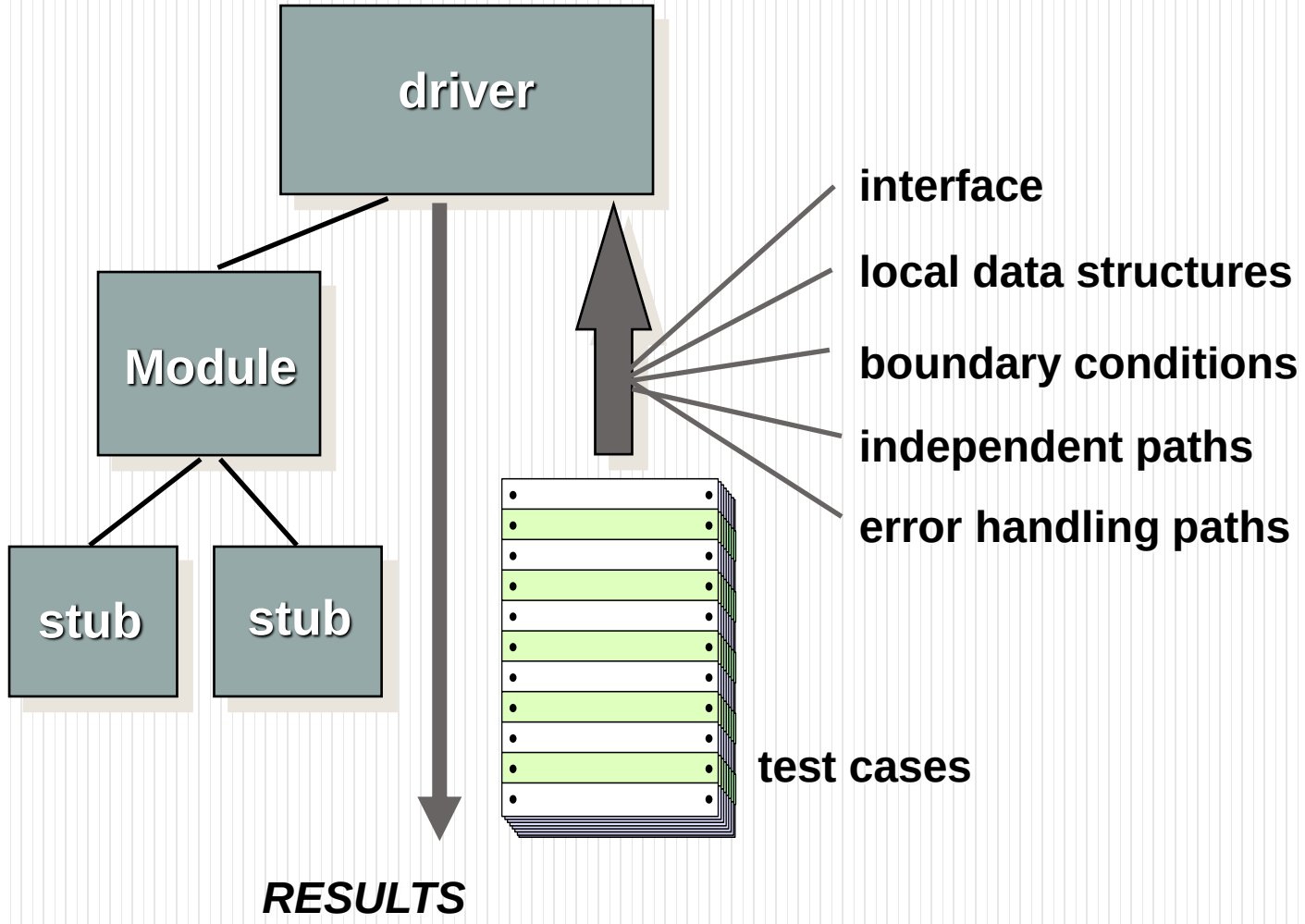


Stubs and drivers

- Driver:
  - A component, that calls the `TestedUnit`
  - Controls the test cases
- Stub:
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.

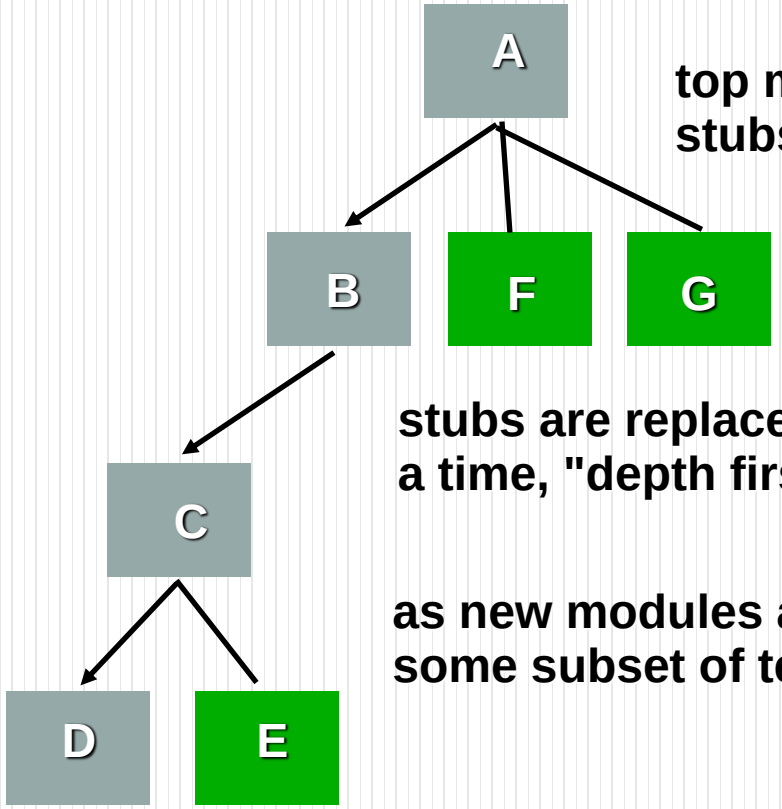


Unit Test Environment



Integration Testing Strategies

Top Down Integration



top module is tested with stubs

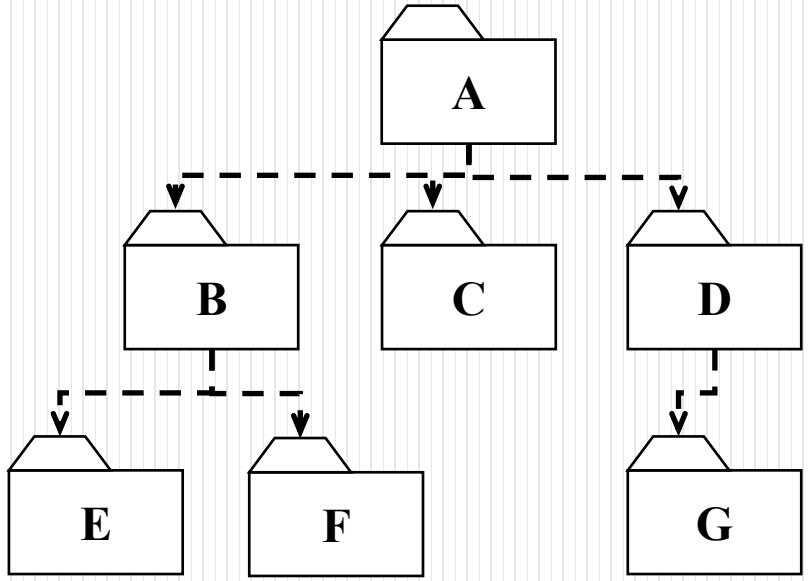
stubs are replaced one at a time, "depth first"

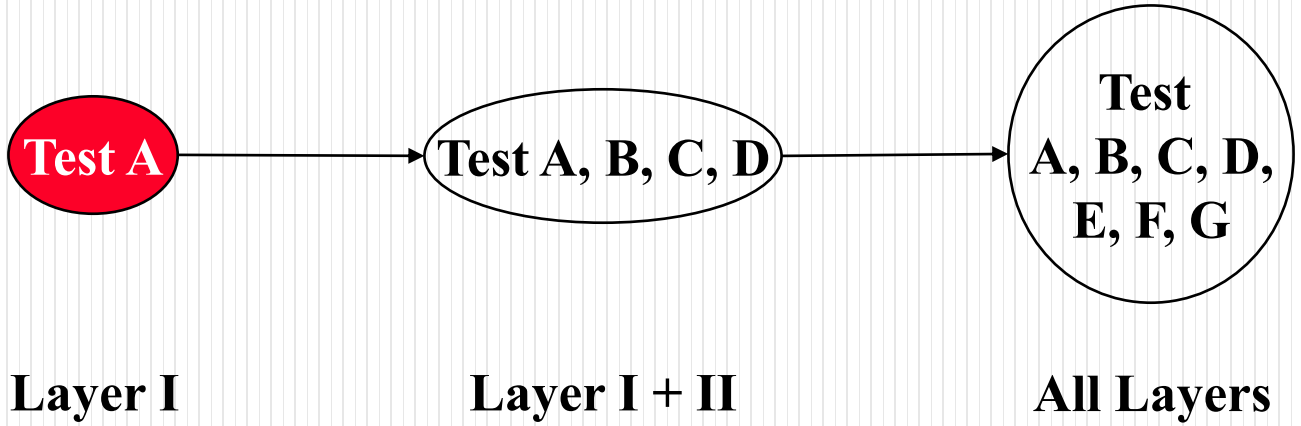
as new modules are integrated, some subset of tests is re-run

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.

Top-down Integration





Pros and Cons of Top-down Integration Testing

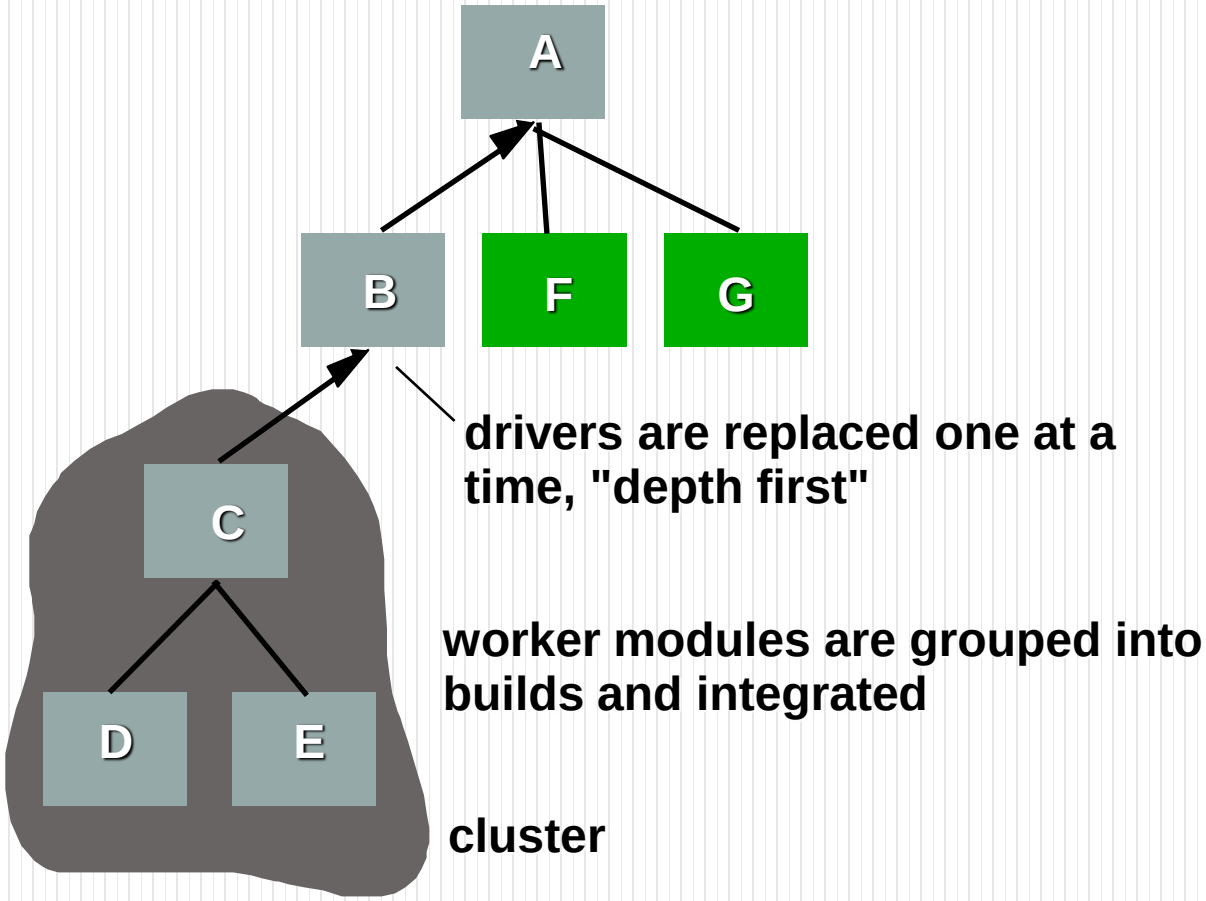
Pro

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

Cons

- Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
- Large number of stubs may be required, especially if the lowest level of the system contains many methods.
- Some interfaces are not tested separately.

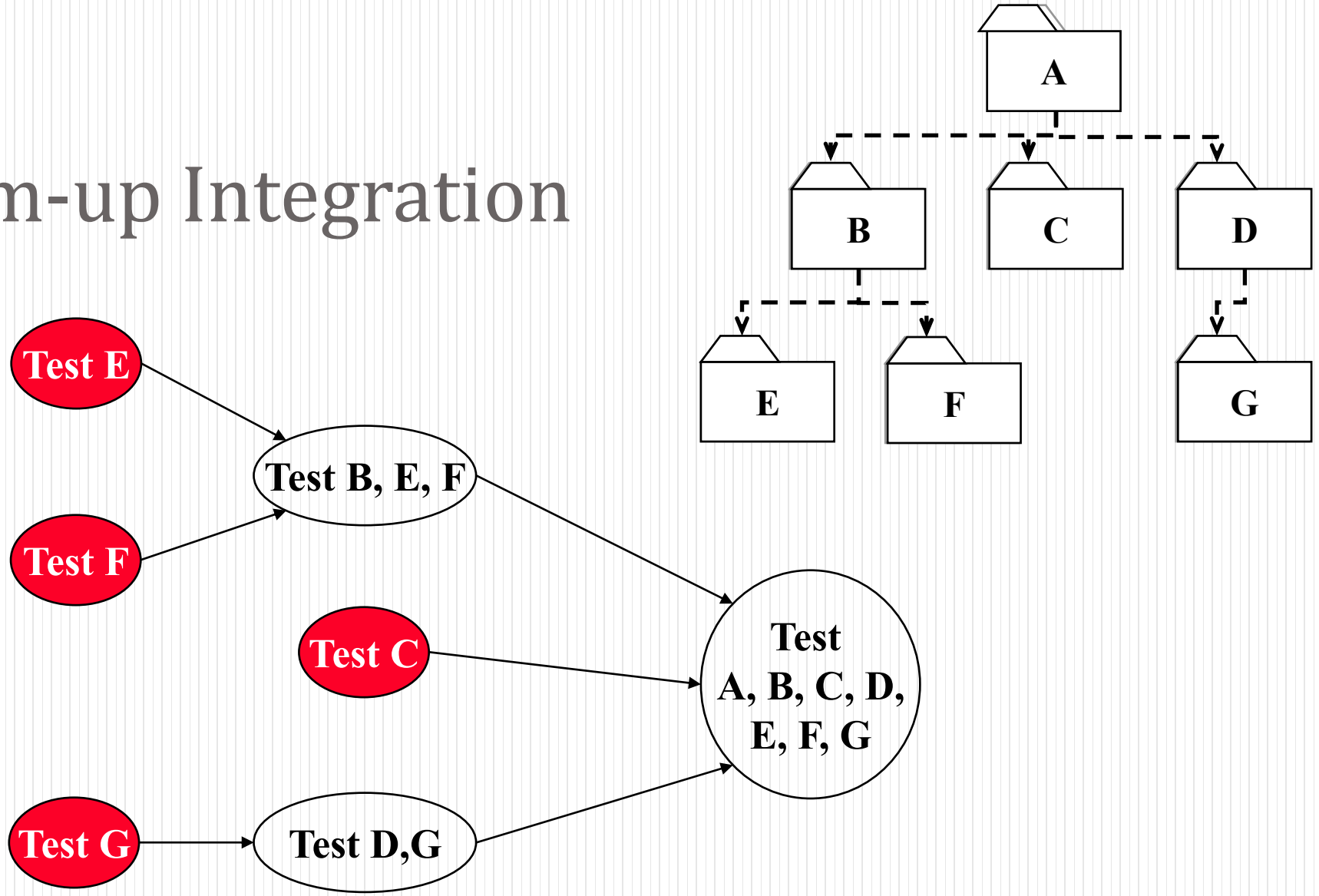
Bottom-Up Integration



Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is repeated until all subsystems are included
- Drivers are needed.

Bottom-up Integration



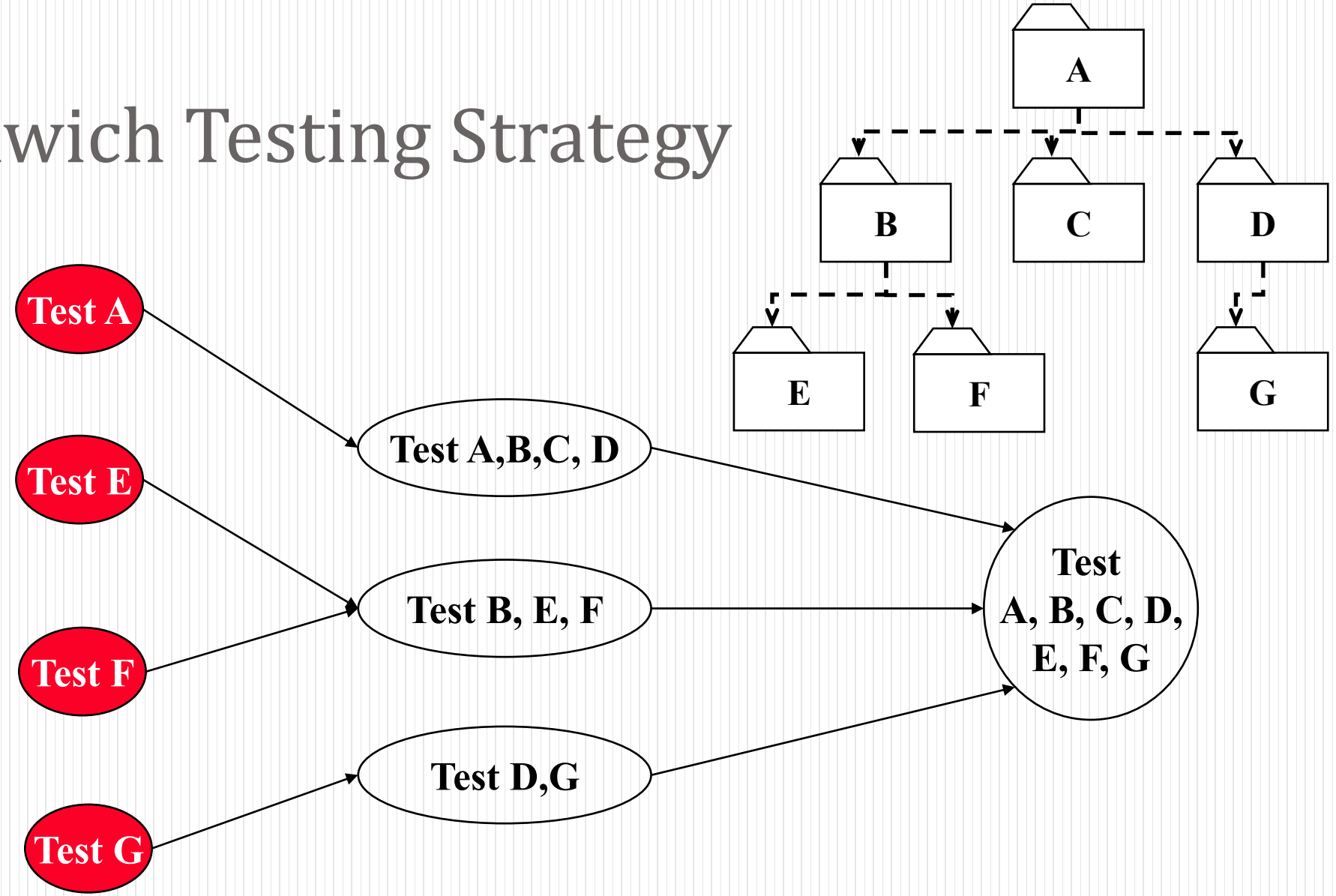
Pros and Cons of Bottom-Up Integration Testing

- Con:
  - Tests the most important subsystem (user interface) last
  - Drivers needed
- Pro
  - No stubs needed

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is viewed as having three layers
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
- Testing converges at the target layer.

Sandwich Testing Strategy



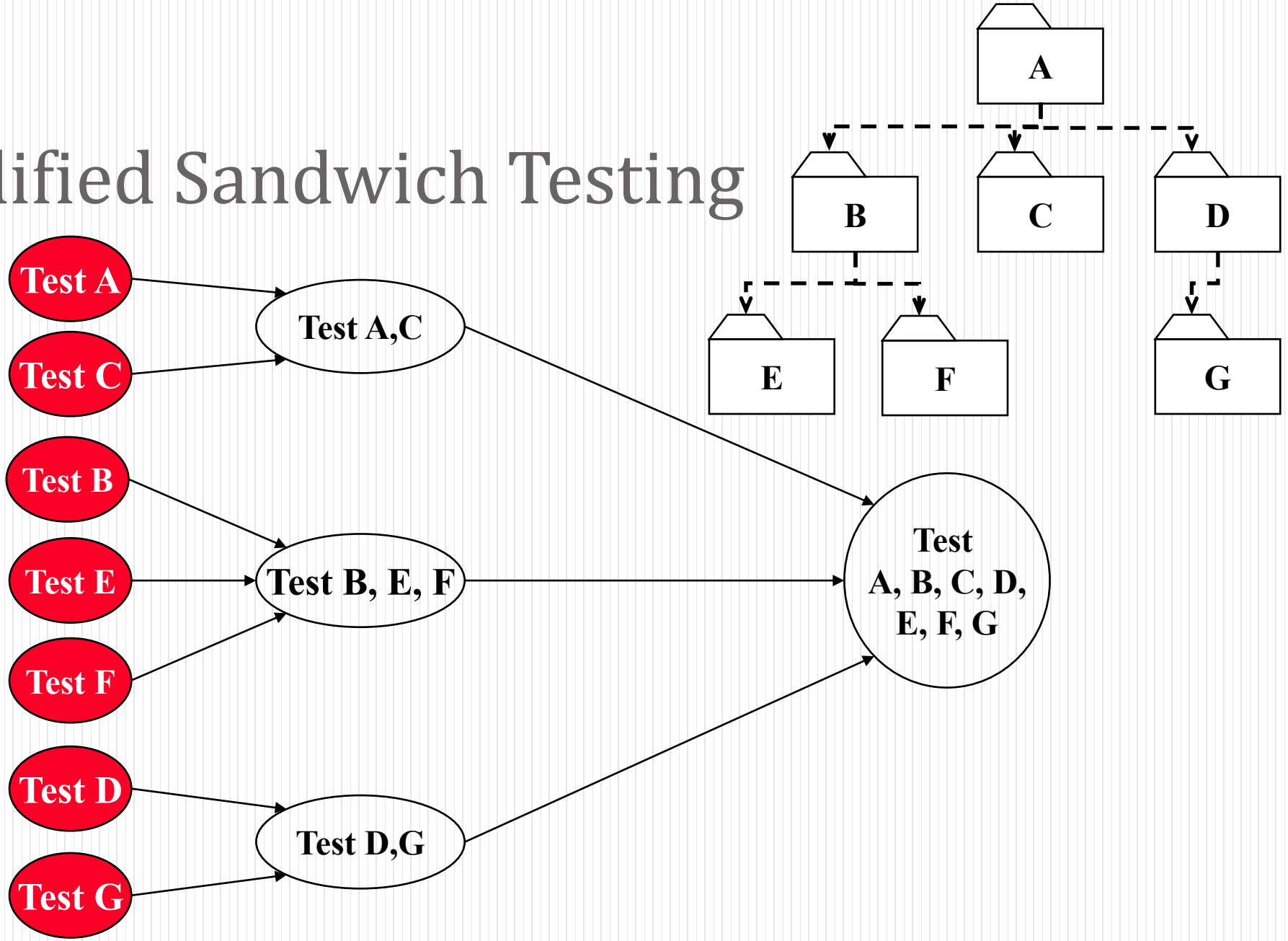
Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Problem: Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy

Modified Sandwich Testing Strategy

- Test in parallel:
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers
- Test in parallel:
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs).

Modified Sandwich Testing



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.

Sanity/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.

Validation testing

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.
- It answers to the question, Are we building the right product?
- Acceptance Testing:
  - Alpha and Beta Testing

Alpha and beta testing

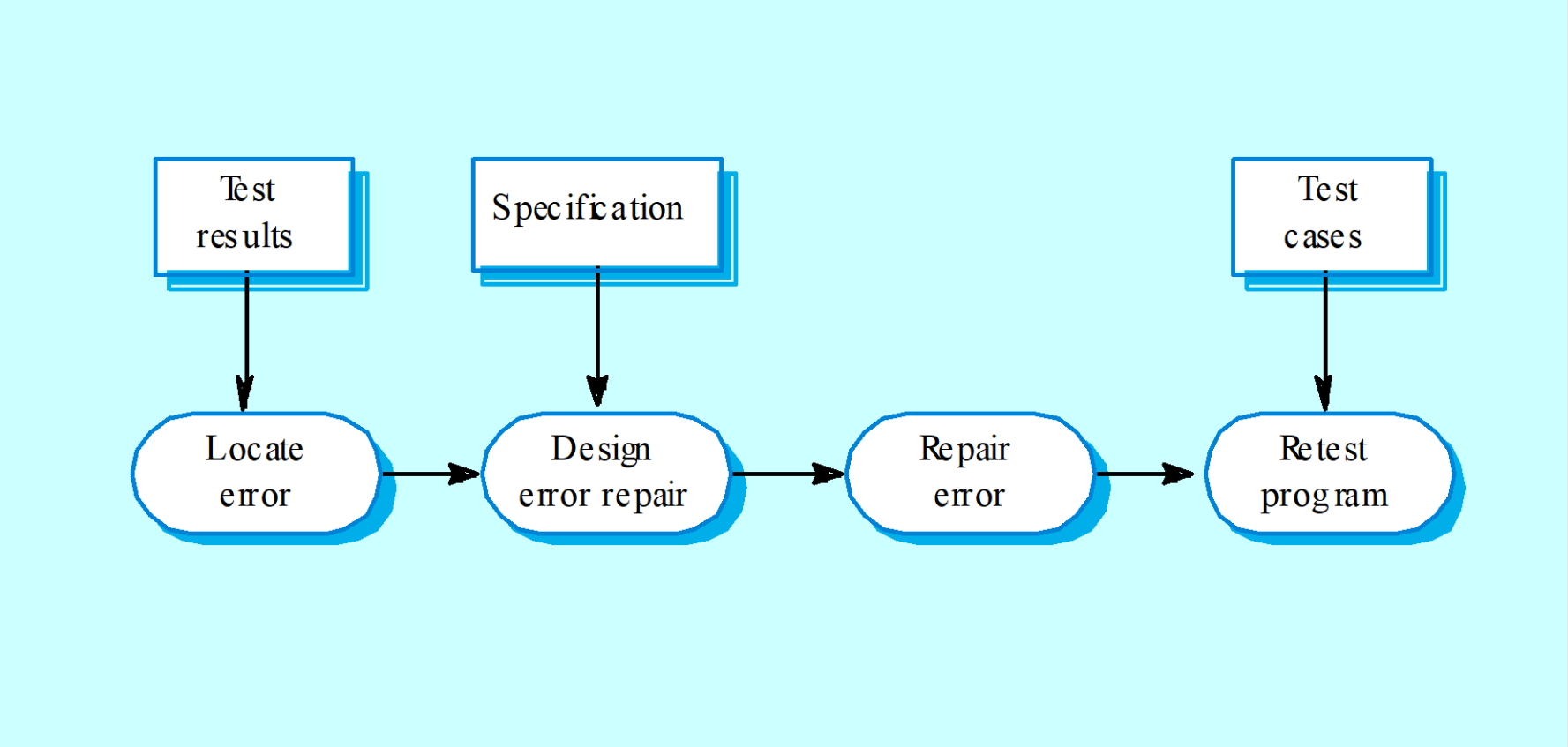
- Acceptance Testing
  - Acceptance testing, a testing technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it is has met the required criteria for delivery to end users.
- Alpha Testing
  - Alpha testing takes place at the developer's site by the internal teams, before release to external customers. This testing is performed without the involvement of the development teams.
- Beta Testing
  - Beta testing also known as user testing takes place at the end users site by the end users to validate the usability, functionality, compatibility, and reliability testing.

System Testing

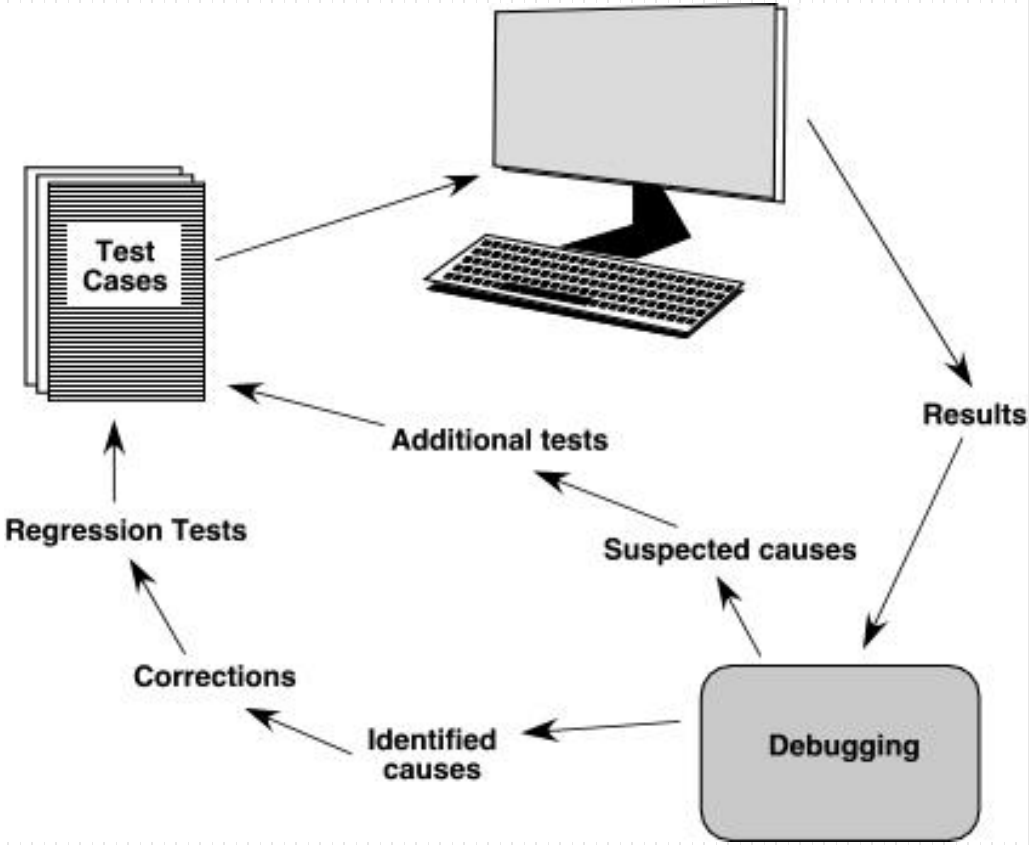
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system
- Deployment Testing
  - software must execute on a variety of platforms and under more than one operating system environment

Debugging: A Diagnostic Process

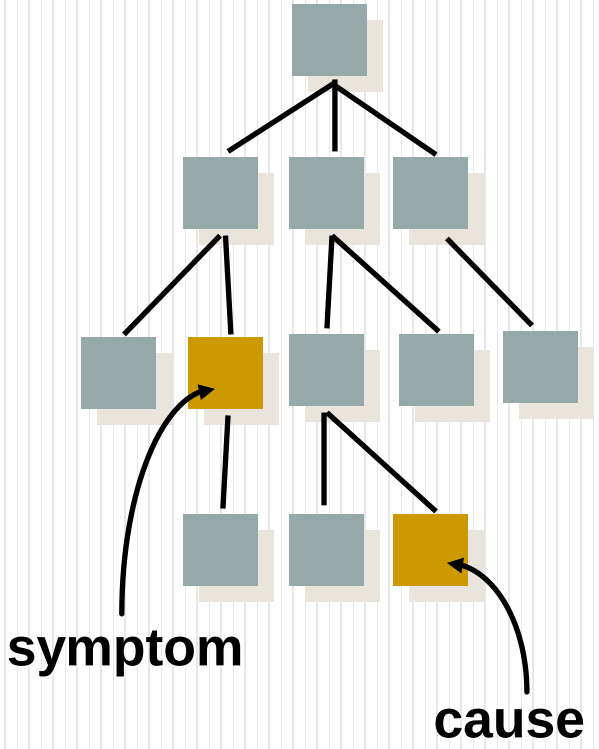
The debugging process



The Debugging Process

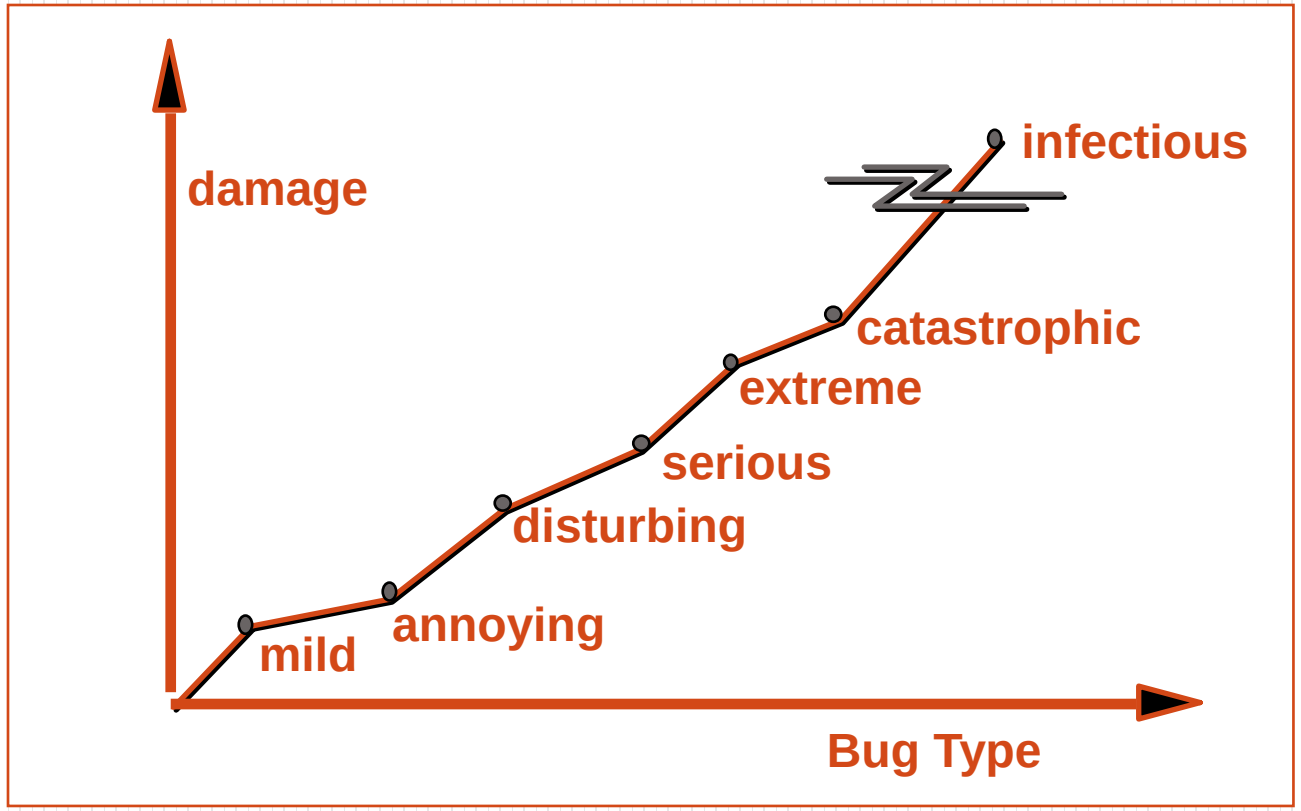


Symptoms & Causes



- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- symptom may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

Consequences of Bugs



Bug Categories: **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

Debugging Techniques

- **Brute force / testing**
- **Backtracking**
- **Cause elimination**

Testing conventional applications

Chapter 18

Roger Pressman – 7th Edition

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

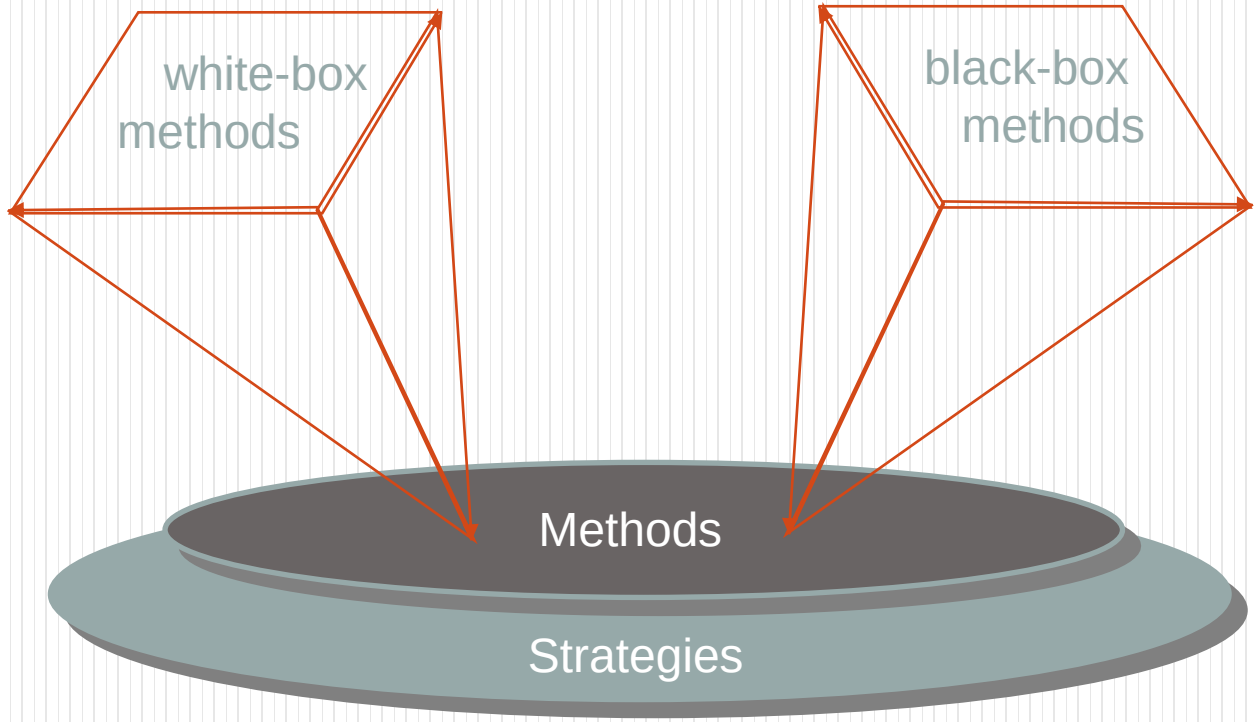
What is a “Good” Test?

- 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

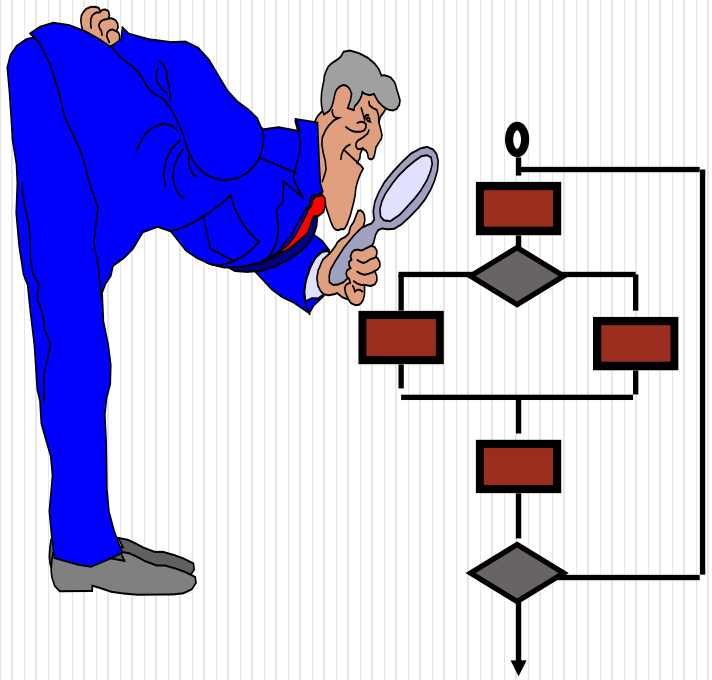
Internal and External Views

- 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; (black box testing)
  - 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. (white box testing)

Software Testing



White-Box Testing

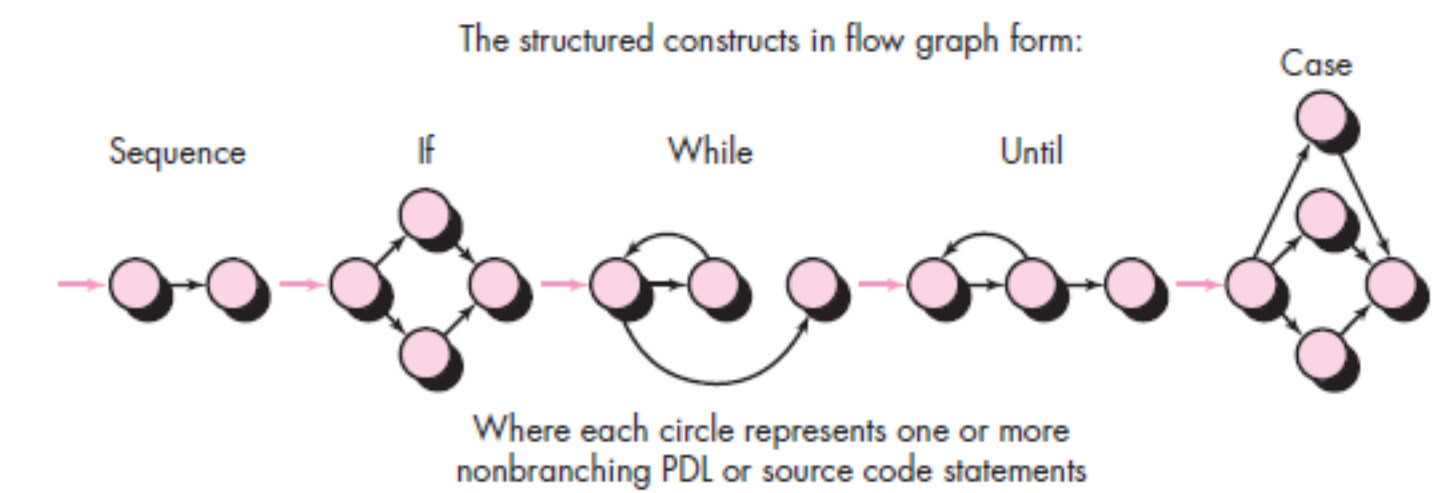


Using white-box testing methods, you can derive test cases that
(1) guarantee that all independent paths within a module have been exercised at least once,
(2) exercise all logical decisions on their true and false sides,
(3) Execute all loops at their boundaries and within their operational bounds, and
(4) Exercise internal data structures to ensure their validity.

Basis path Testing

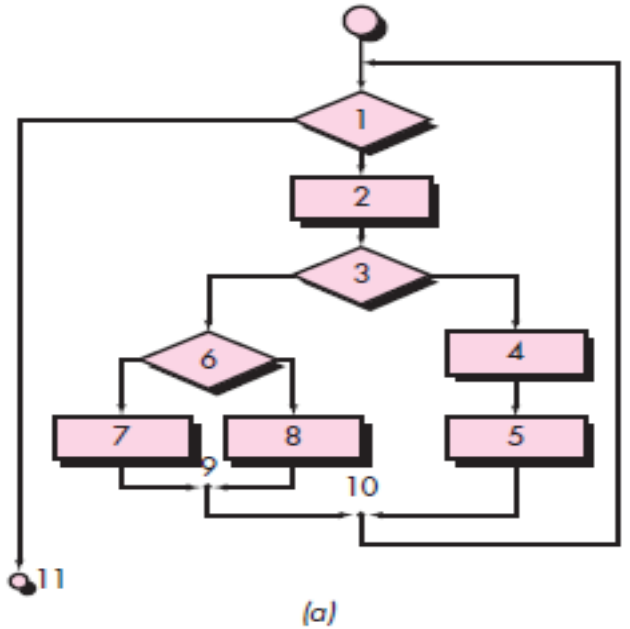
- **Basis path testing**, or structured **testing**, is a white box method for designing test cases. The method analyzes the control flow graph of a program to find a set of linearly independent **paths** of execution.
- The method analyzes the control flow graph of a program to find a set of linearly independent paths of execution.
- The method normally uses cyclomatic complexity to determine the number of linearly independent paths and then generates test cases for each path thus obtained.
- Basis path testing guarantees complete branch coverage (all CFG edges), but achieves that without covering all possible CFG paths—the latter is usually too costly.

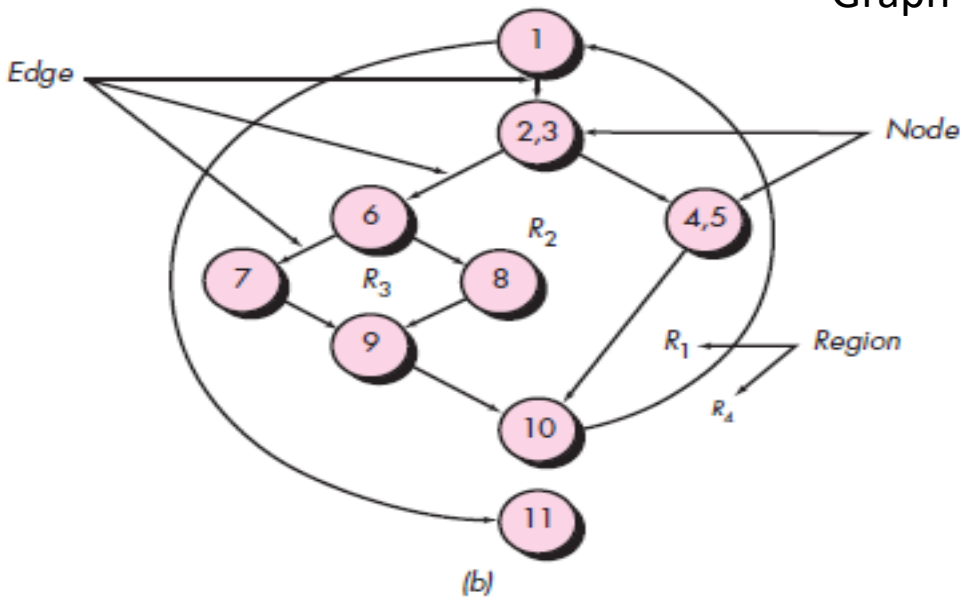
Control flow graph



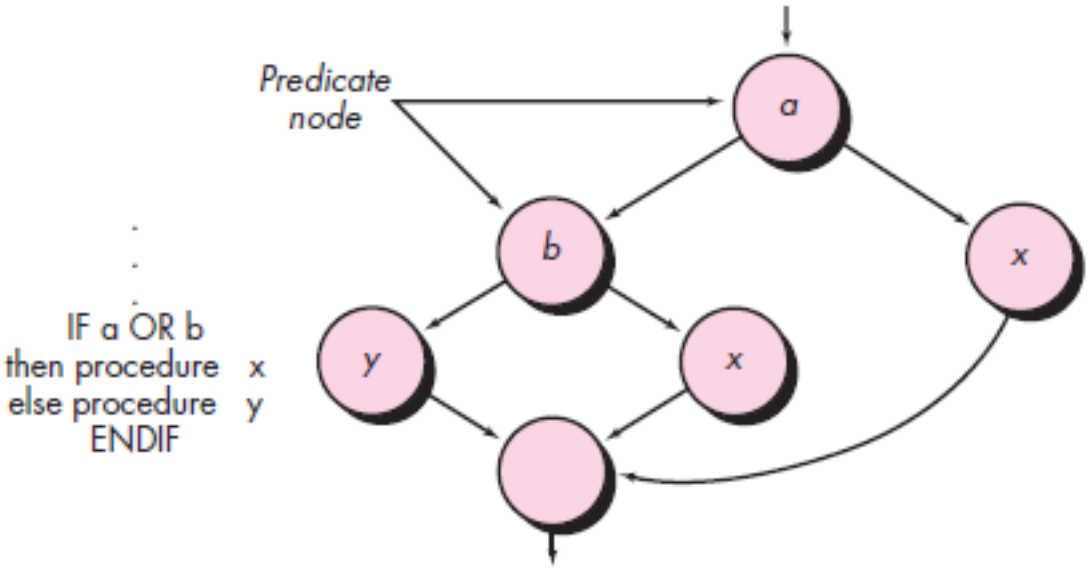
Flow graph notation

a. Flowchart
b. Control Flow Graph





Predicate nodes



Compound Logic

- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.
- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
- The program design language (PDL) segment translates into the flow graph shown.
- Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

Independent program paths

- An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.
- When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.
- 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
- Paths 1 through 4 constitute a *basis set* for the flow graph.
- That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.
- How do you know how many paths to look for?

Cyclomatic complexity

- *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- Complexity is computed in one of three ways:

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

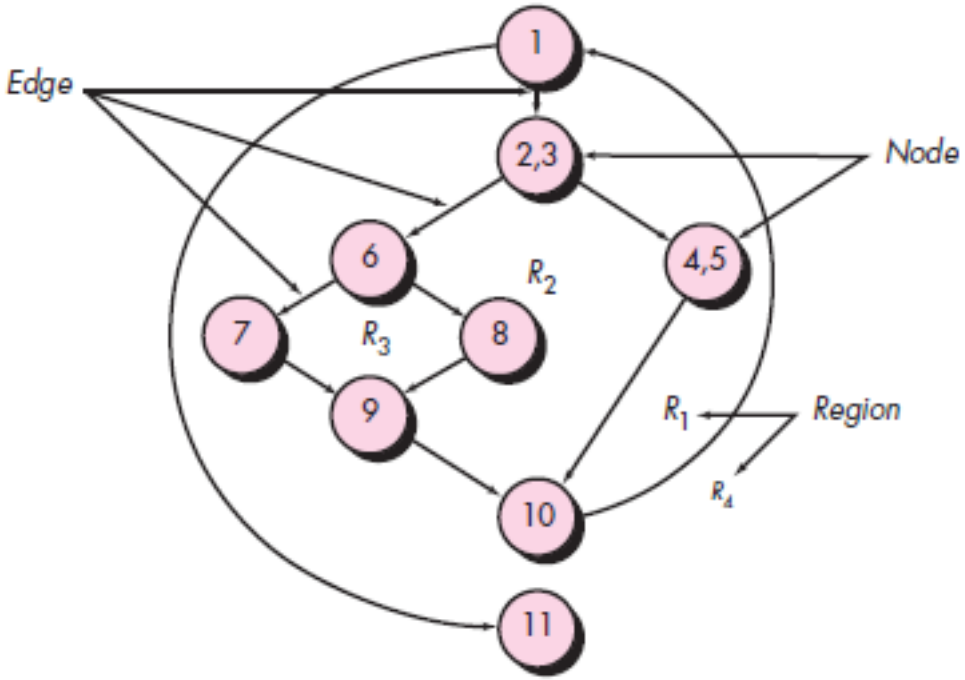
$V(G) = E - N + 2$ where E is the number of flow graph edges and N is the number of flow graph nodes.

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

$V(G) = P + 1$ where P is the number of predicate nodes contained in the flow graph G.

Cyclomatic complexity

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



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.

Exercise

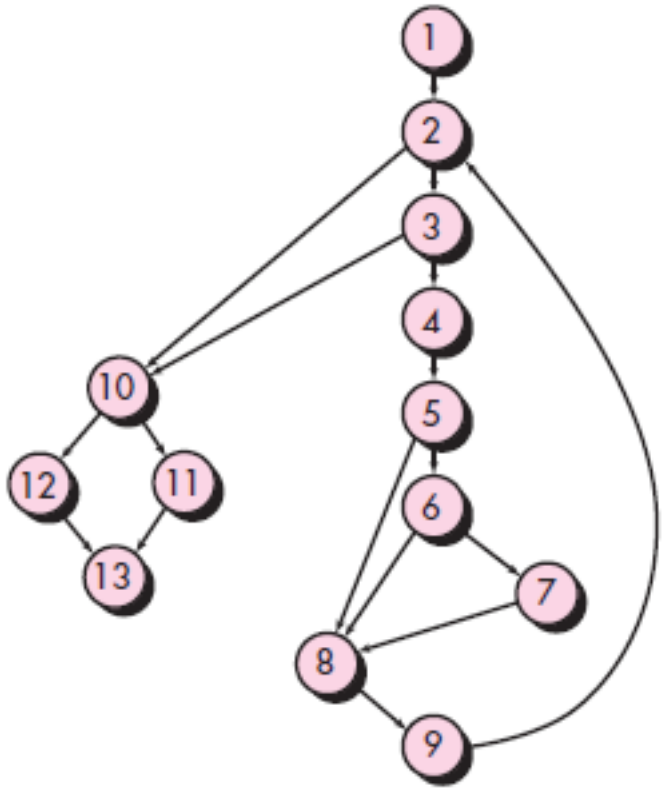
```
PROCEDURE average;

  *  This procedure computes the average of 100 or fewer
     numbers that lie between bounding values; it also computes the
     sum and the total number valid.

  INTERFACE RETURNS average, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;

  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
1 i = 1;
  total.input = total.valid = 0;                                  2
  sum = 0;
  DO WHILE value[i] <> -999 AND total.input < 100                 3
4   increment total.input by 1;
    IF value[i] > = minimum AND value[i] < = maximum              6
5     THEN increment total.valid by 1;
7              sum = s sum + value[i]
      ELSE skip
8   ENDIF
    increment i by 1;
9 ENDDO
  IF total.valid > 0                                              10
11  THEN average = sum / total.valid;
12    ELSE average = -999;
13 ENDIF
END average
```

Solution

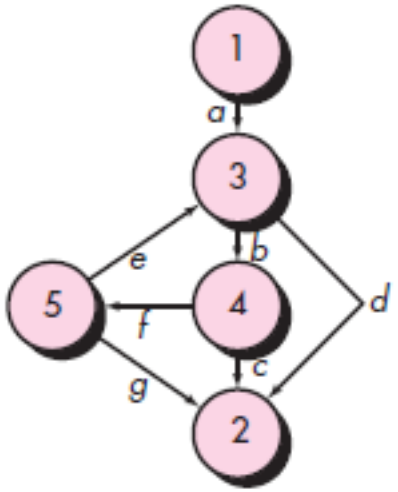


V(*G*) = 6 regions
V(*G*) = 17 edges - 13 nodes + 2 = 6
V(*G*) = 5 predicate nodes + 1 = 6

Path 1: 1-2-10-11-13
Path 2: 1-2-10-12-13
Path 3: 1-2-3-10-11-13
Path 4: 1-2-3-4-5-8-9-2-. . .
Path 5: 1-2-3-4-5-6-8-9-2-. . .
Path 6: 1-2-3-4-5-6-7-8-9-2-. . .
The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

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

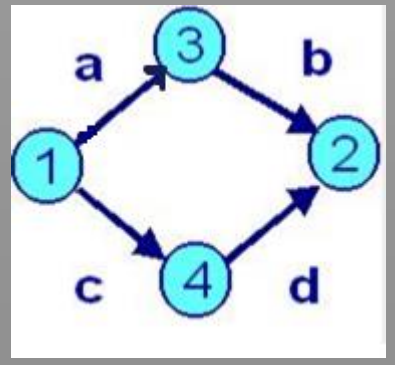


Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices



	1	2	3	4
1			a	c
2				
3		b		
4		d		

	1	2	3	4
1			1	1
2				
3		1		
4		1		

Connection Matrix

1			1	1
2				
3		1		
4		1		

2 - 1 = 1

1 - 1 = 0

1 - 1 = 0

1 + 1 = 2 = V(G)

Calculation of V(G)

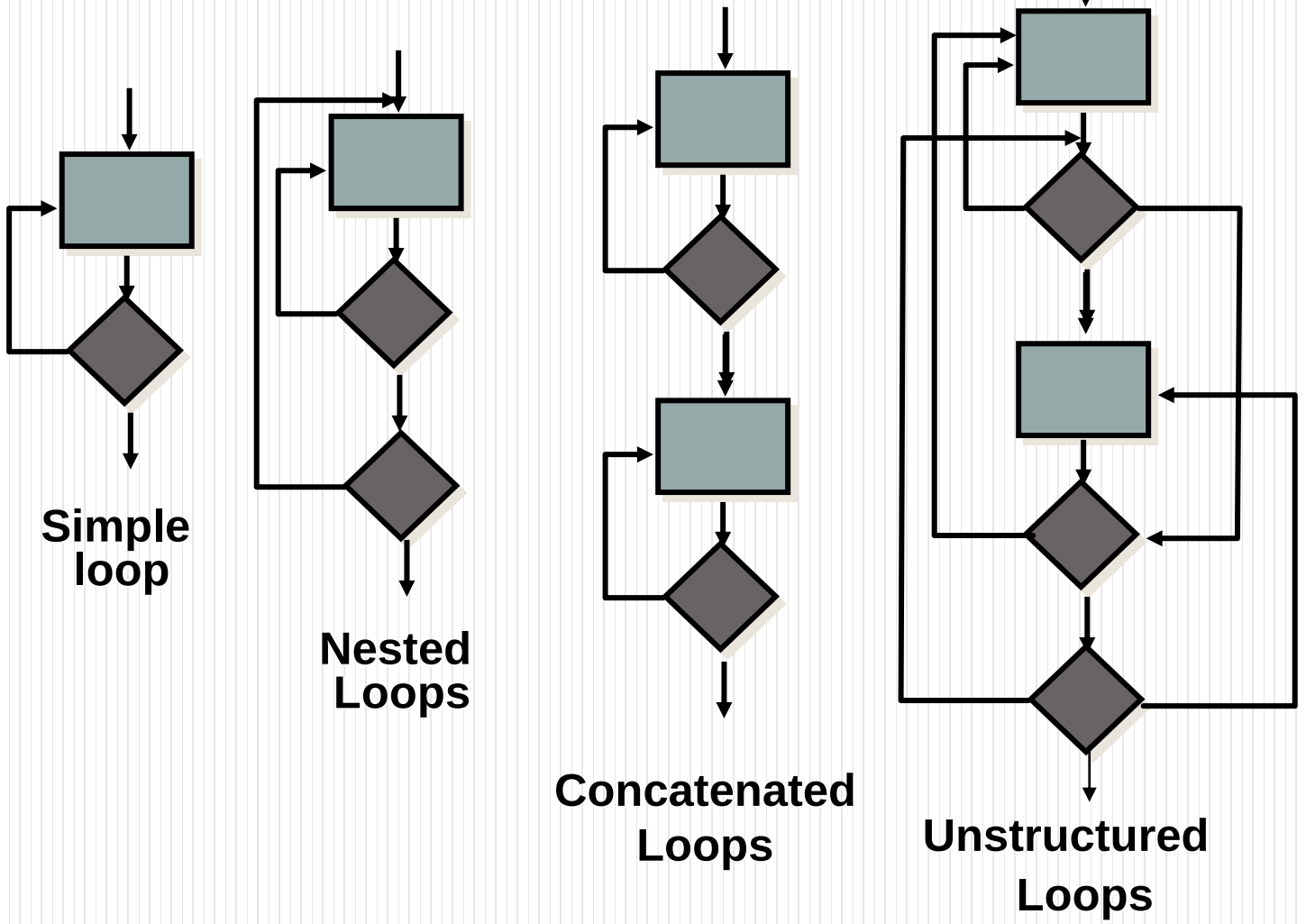
Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program
- Loop testing – can define loops as simple, concatenated, nested and unstructured.

Data Flow Testing

- The data flow testing method 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′*

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

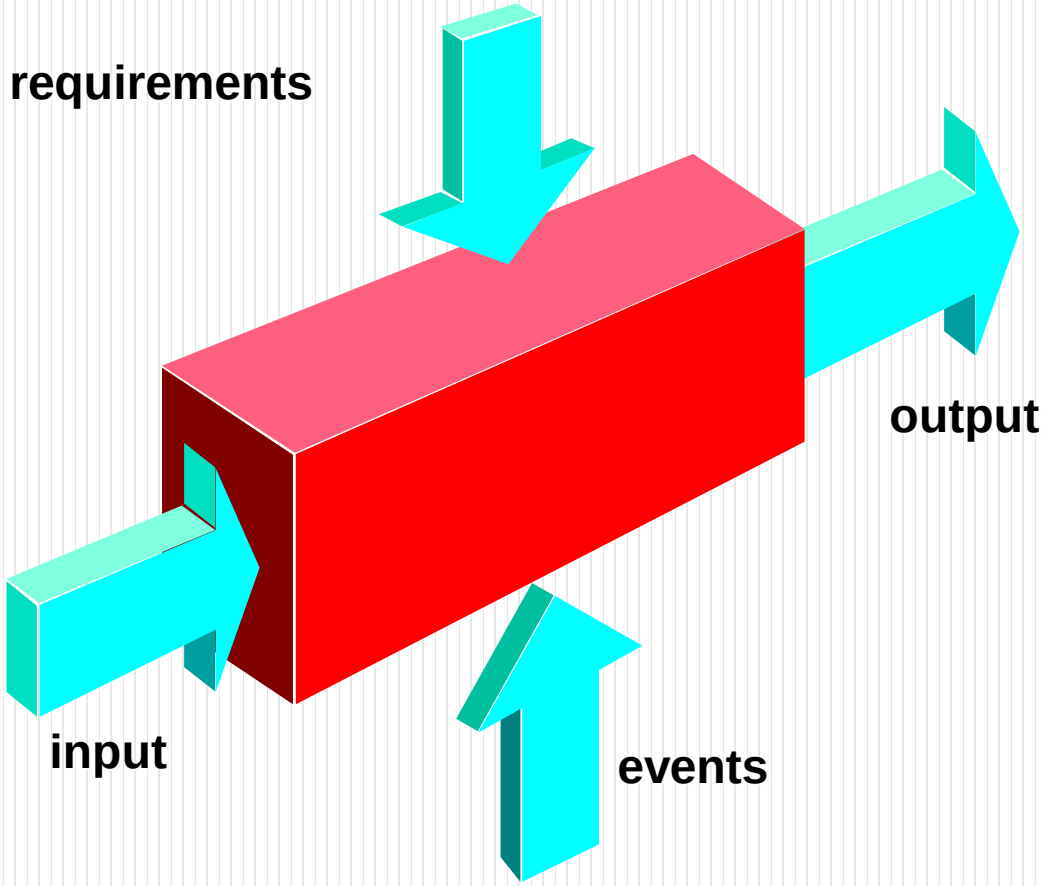
If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

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

White Box Testing

Advantages	Disadvantages
•As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively. •It helps in optimizing the code. •Extra lines of code can be removed which can bring in hidden defects. •Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	•Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased. •Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested. •It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.

Black-Box Testing



Black-Box Testing

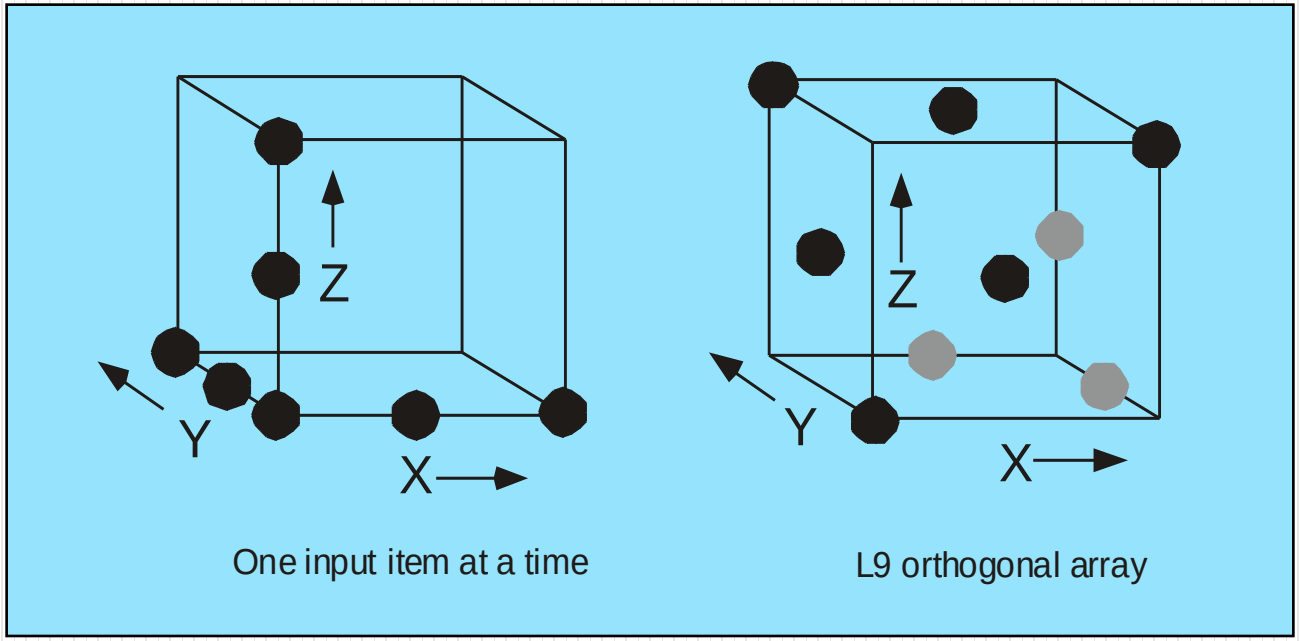
- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Equivalence class partitioning and boundary value analysis

- Refer to book as well as below link for the same:
  - http://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html

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



Orthogonal Array Testing

- Refer to book as well as below link for the same:
  - http://www.softwaretestinghelp.com/combinational-test-technique/

Black Box Testing

Advantages	Disadvantages
•Well suited and efficient for large code segments. •Code access is not required. •Clearly separates user's perspective from the developer's perspective through visibly defined roles. •Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	•Limited coverage, since only a selected number of test scenarios is actually performed. •Inefficient testing, due to the fact that the tester only has limited knowledge about an application. •Blind coverage, since the tester cannot target specific code segments or error-prone areas. •The test cases are difficult to design.

Comparison

Black-Box Testing	White-Box Testing
The internal workings of an application need not be known.	Tester has full knowledge of the internal workings of the application.
Also known as closed-box testing, data-driven testing, or functional testing.	Also known as clear-box testing, structural testing, or code-based testing.
Performed by end-users and also by testers and developers.	Normally done by testers and developers.
Testing is based on external expectations - Internal behavior of the application is unknown.	Internal workings are fully known and the tester can design test data accordingly.
It is exhaustive and the least time-consuming.	The most exhaustive and time-consuming type of testing.
Not suited for algorithm testing.	Suited for algorithm testing.
This can only be done by trial-and-error method.	Data domains and internal boundaries can be better tested.

Testing object-oriented applications

Chapter 19

Roger Pressman – 7th Edition

OO Testing

- 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, and
  - the design of test cases must account for the unique characteristics of OO software.

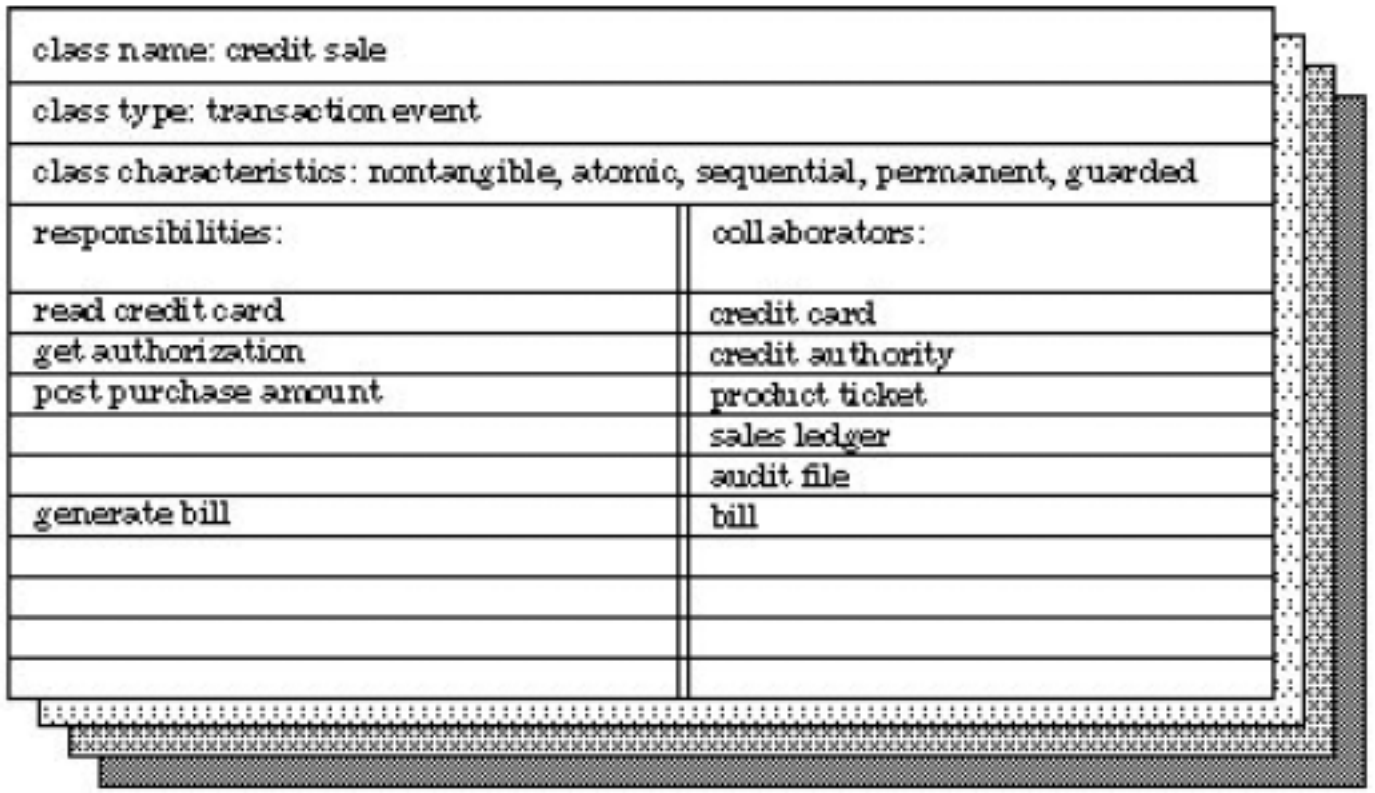
‘Testing’ OO Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side affects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Class Model Consistency

- Revisit the CRC model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- Determine whether widely requested responsibilities might be combined into a single responsibility.

Class Model Consistency



class name: credit sale	
class type: transaction event	
class characteristics: nontangible, atomic, sequential, permanent, guarded	
responsibilities:	collaborators:
read credit card	credit card
get authorization	credit authority
post purchase amount	product ticket
	sales ledger
	audit file
generate bill	bill

OO Testing Strategies

- Unit testing (Class 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
  - the validation of OO software focuses on user-visible actions and user-recognizable outputs from the system
  - draw upon use cases that are part of the requirements model

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.
- 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.

OOT Methods at class level

- Random Testing
- Partition Testing
- Inter class Testing
  - Multiple class testing
  - Test derived from behavior models

OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a minimum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

OOT Methods: Random Testing

- Consider a banking application in which an **Account** class has the following operations: *open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(),* and *close()*. Each of these operations may be applied for **Account,** but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of **Account** includes the following operations:
  - *open•setup•deposit•withdraw•close*
- This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:
  - *open•setup•deposit•[deposit|withdraw|balance|summarize|creditLimit]n•withdraw•close*
- A variety of different operation sequences can be generated randomly. For example:
  - *Test case r1: open•setup•deposit•deposit•balance•summarize•withdraw•close*
  - *Test case r2: open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close*

OOT Methods: Partition Testing

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

OOT Methods: Partition Testing

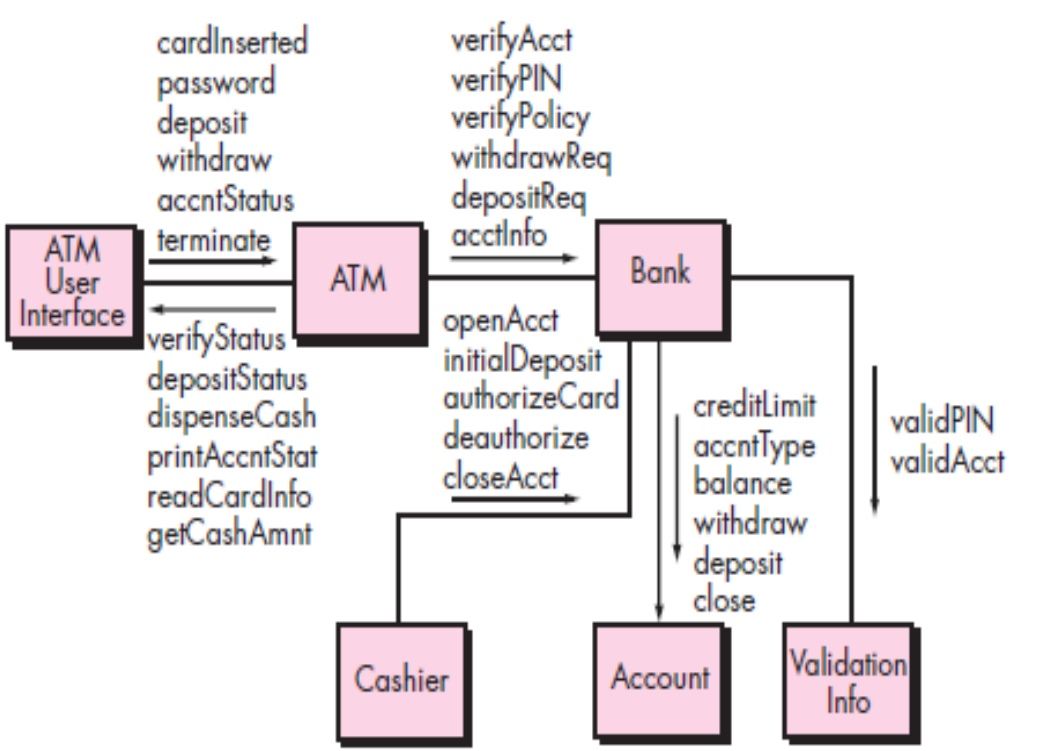
- *State-based partitioning : Co*nsidering the **Account** class, state operations include *deposit()* and *withdraw(),* whereas nonstate operations include *balance(), summarize(),* and *creditLimit().* Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,
  - *Test case p1: open•setup•deposit•deposit•withdraw•withdraw•close*
  - *Test case p2: open•setup•summarize•creditLimit•close*
- Test case *p*1 changes state, while test case *p*2 exercises operations that do not change state (other than those in the minimum test sequence).
- *Attribute-based partitioning :* For the **Account** class, the attributes *balance* and *creditLimit* can be used to define partitions. Operations are divided into three partitions: (1) operations that use *creditLimit,* (2) operations that modify *creditLimit*, and (3) operations that do not use or modify *creditLimit.* Test sequences are then designed for each partition.
- *Category-based partitioning* : For example, operations in the **Account** class can be categorized in initialization operations (*open, setup*), computational operations (*deposit, withdraw*), queries (*balance, summarize, creditLimit*), and termination operations (*close*).

OOT Methods: Inter-Class Testing

- Multiple class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

OOT Methods: multiple Class Testing

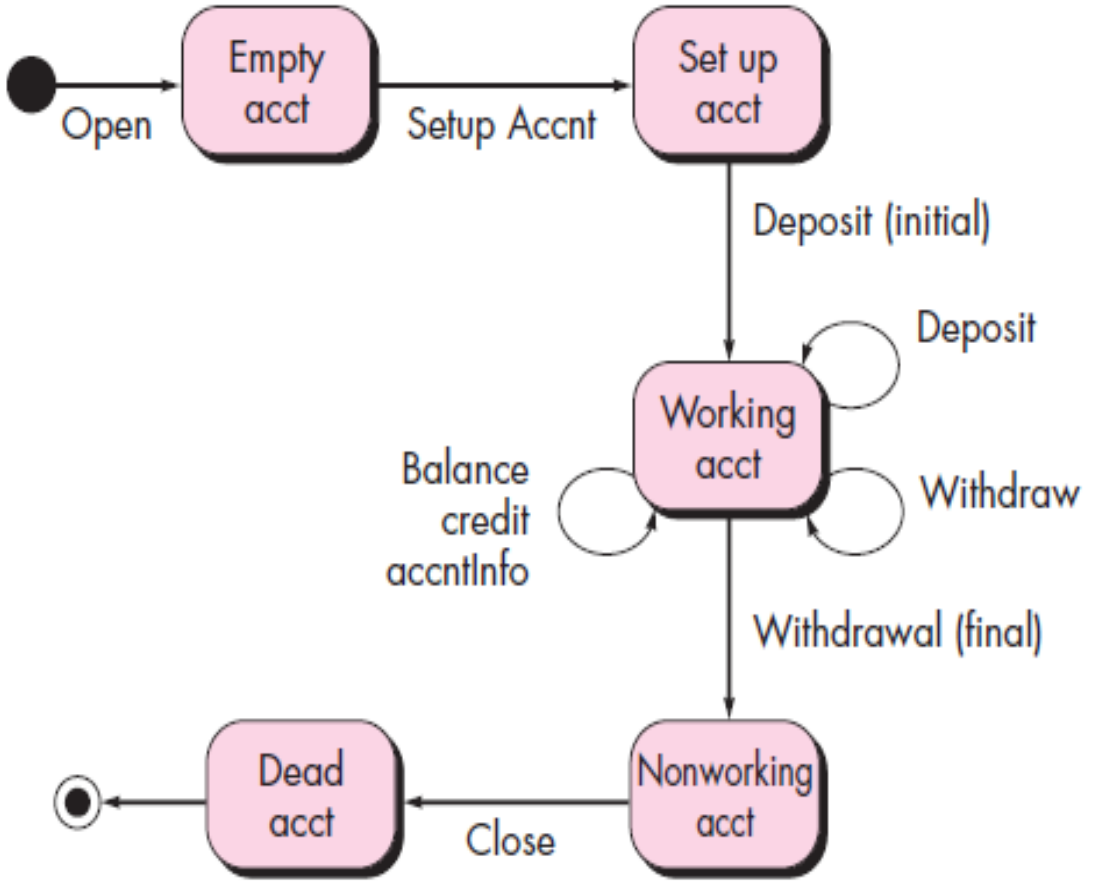
- To illustrate , consider a sequence of operations for the **Bank** class relative to an **ATM** class:
- *verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]^n*
- A random test case for the **Bank** class might be
- *Test case r3 verifyAcct•verifyPIN•depositReq*
- In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case *r3* are considered. **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct()* and *verifyPIN().* **Bank** must collaborate with **Account** to execute *depositReq().* Hence, a new test case that exercisesthese collaborations is
- *Test case r4*
- *verifyAcct[Bank:validAcctValidationInfo]•verifyPIN[Bank: validPinValidationInfo]•depositReq [Bank: depositaccount]*



OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage.

That is, the operation sequences should cause the Account class to make transition through all allowable states



Test case s1: open•setupAccnt•deposit (initial)•withdraw (final)•close
Test case s2: open•setupAccnt•deposit(initial)•deposit•balance• credit•withdraw (final)•close
Test case s3: open•setupAccnt•deposit(initial)•deposit•withdraw•a ccntInfo•withdraw (final)•close

Verification and Validation

Chapter 22

Sommerville – 8th Edition

Verification vs validation

- Verification:

  "Are we building the product right".

- The software should conform to its specification.
- Validation:

  "Are we building the right product".

- The software should do what the user really requires.

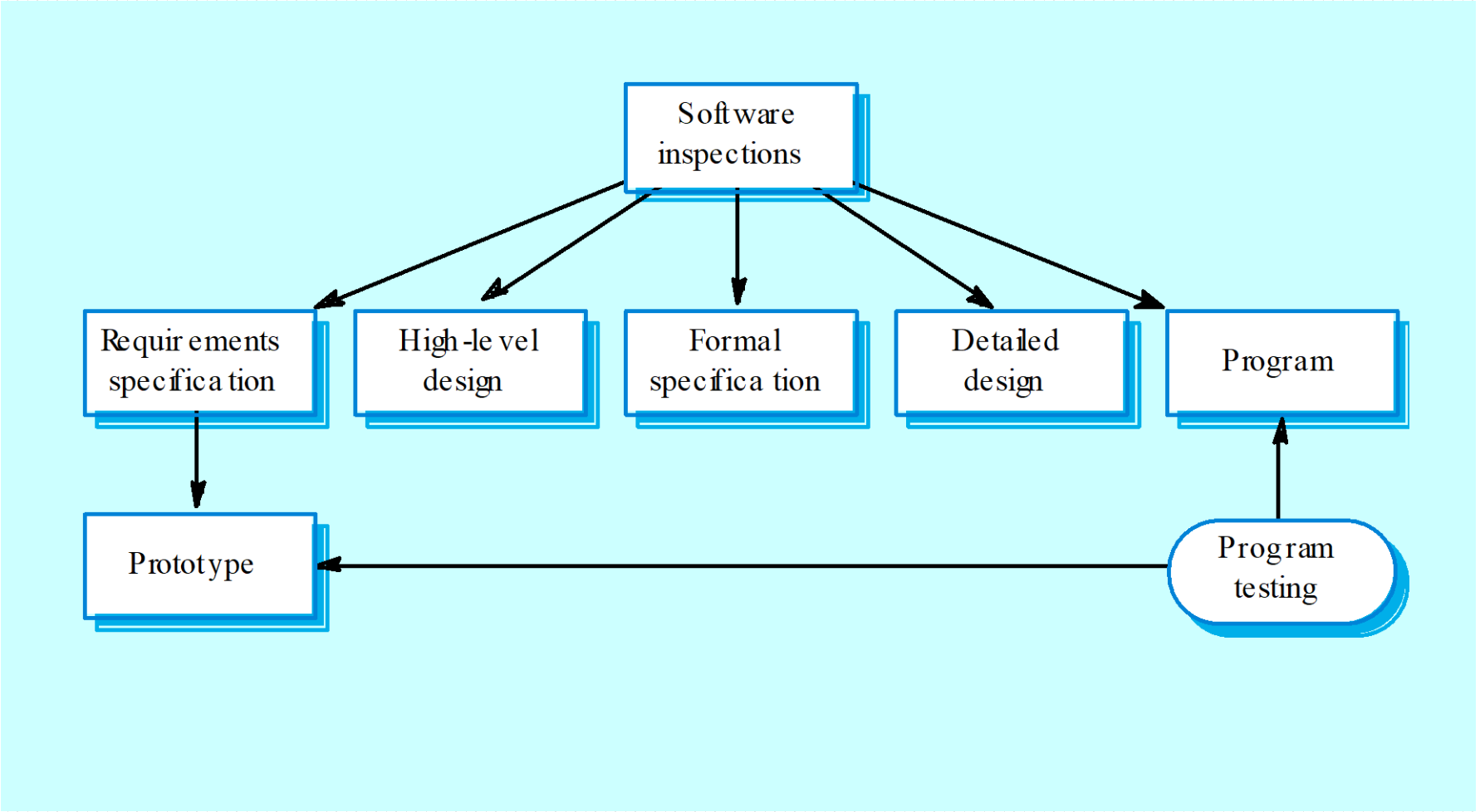
The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system;
  - The assessment of whether or not the system is useful and useable in an operational situation.
- Verification and validation should establish confidence that the software is fit for purpose.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

Static and dynamic verification

- Software inspections. Concerned with analysis of the static system representation to discover problems *(*static verification)
  - May be supplement by tool-based document and code analysis
- Software testing. Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

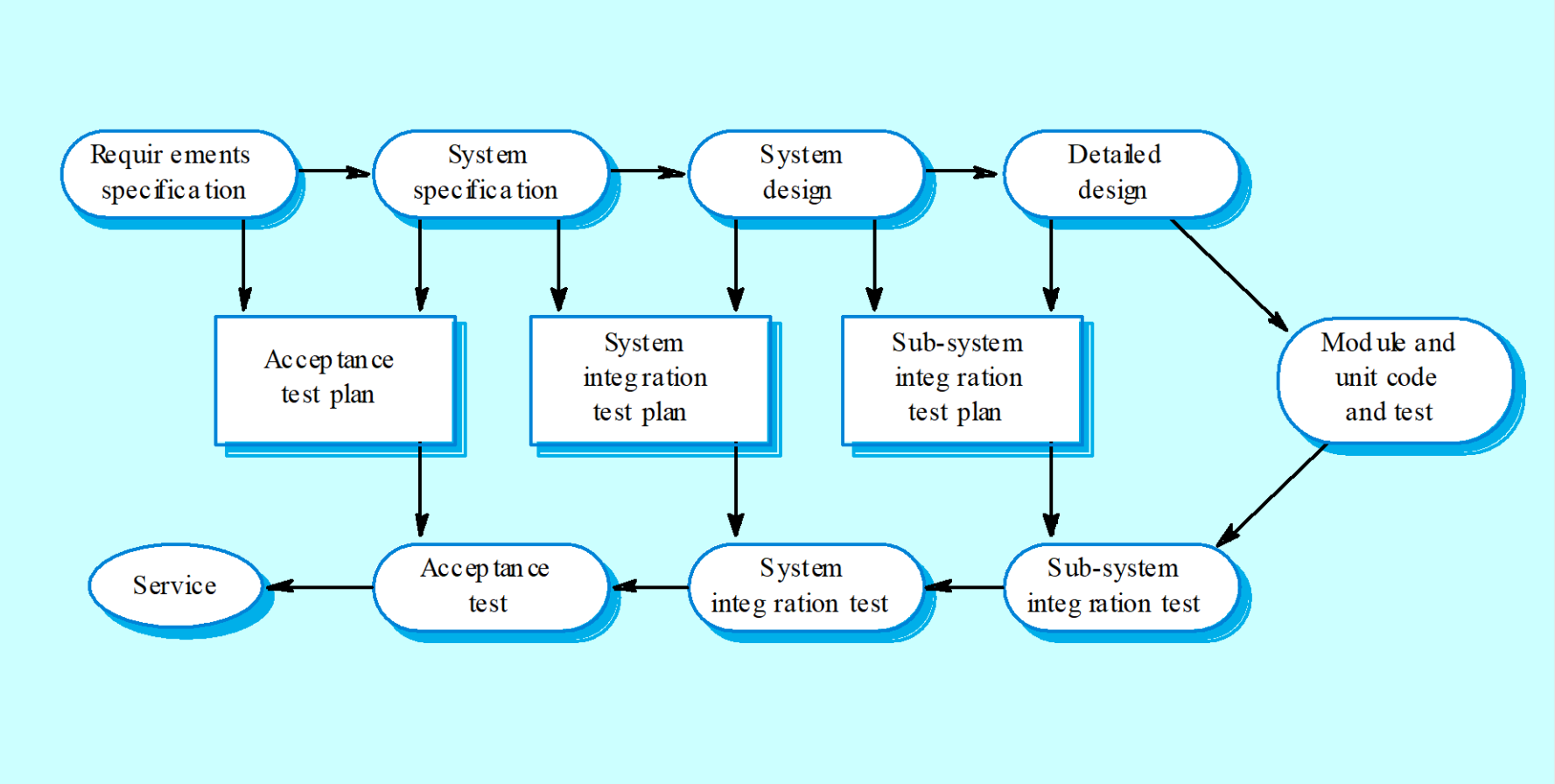
Static and dynamic V&V



V & V planning

- Careful planning is required to get the most out of testing and inspection processes.
- Planning should start early in the development process.
- The plan should identify the balance between static verification and testing.
- Test planning is about defining standards for the testing process rather than describing product tests.

The V-model of development



The structure of a software test plan

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

The software test plan

The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process that are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule is, obviously, linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests; the results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

Hardware and software requirements
This section should set out the software tools required and estimated hardware utilisation.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

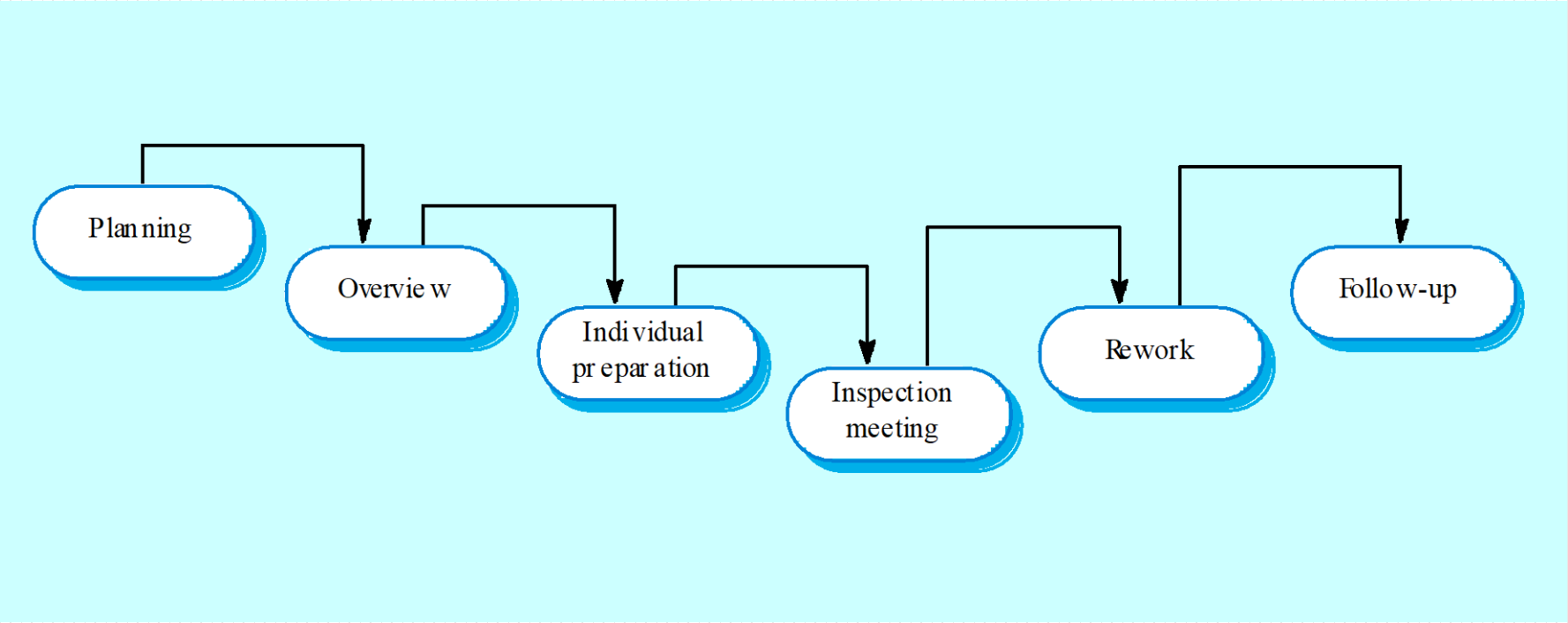
Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.
- Formalised approach to document reviews
- Intended explicitly for defect detection (not correction).

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

The inspection process



Inspection procedure

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair discovered errors.
- Re-inspection may or may not be required.

Inspection roles

Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks 1

Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a de limiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?

Inspection checks 2

Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Stages of static analysis

- Control flow analysis. Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- Data use analysis. Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- Interface analysis. Checks the consistency of routine and procedure declarations and their use

Stages of static analysis

- Information flow analysis. Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- Path analysis. Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. They must be used with care.

Use of static analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler,
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

Thank you!!!

ANY QUESTIONS???