UNIT IV -SOFTWARE TESTING

Complied by
Dept of CSE, SRM IST
,Vadapalani

LIST OF TOPICS

SOFTWARE TESTING

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

Basic Definitions

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer.

Faults (Defects)

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Failures

A failure is the inability of a software system or component to perform its required functions within specified performance requirements [2].

Test Case

A test case in a practical sense is a test-related item which contains the following information:

1. A set of test inputs. These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
2. Execution conditions. These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
3. Expected outputs. These are the specified results to be produced by the code under test.

Test

A test is a group of related test cases, or a group of related test cases and test procedures

Test Bed

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.

VERIFICATION VS VALIDATION

Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance

Verification (Are the algorithms coded correctly?)

- The set of activities that ensure that software correctly implements a specific function or algorithm
- It refers to the set of tasks that ensure that software correctly implements a specific function

Verification: "Are we building the product right?"

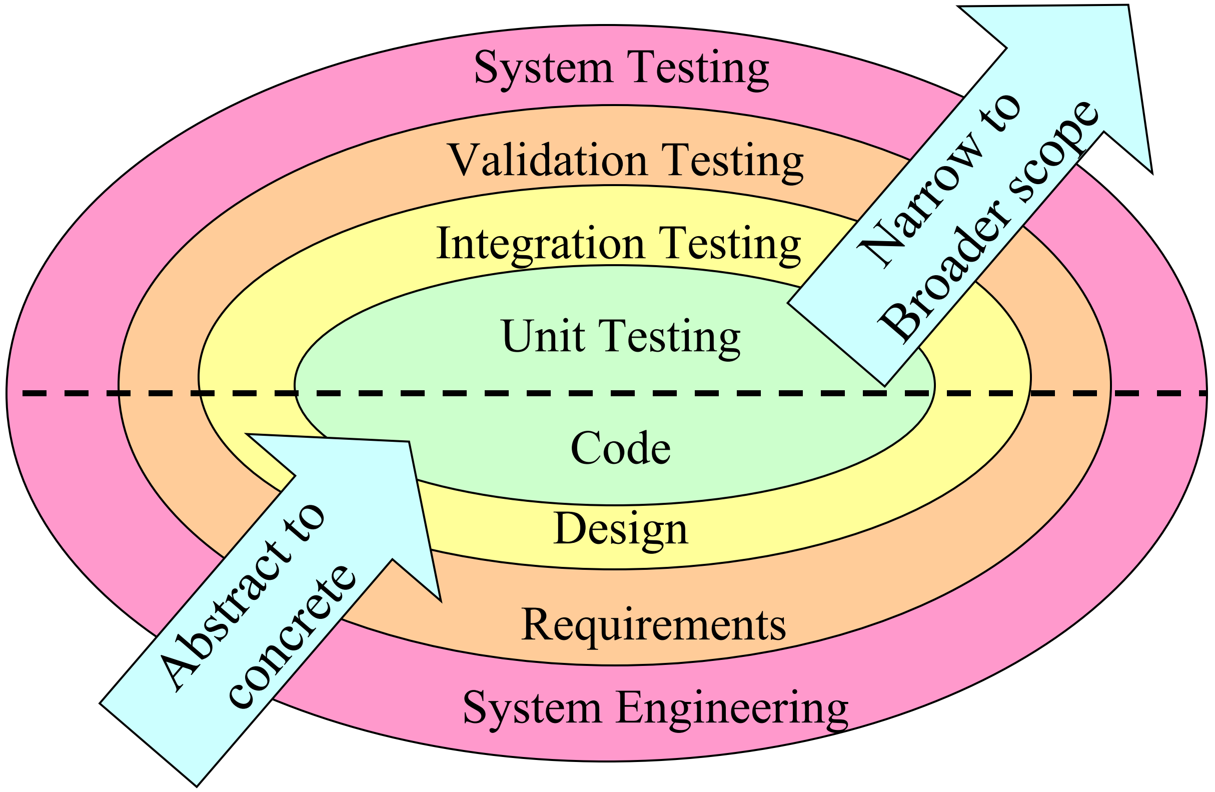
Validation (Does it meet user requirements?)

- The set of activities that ensure that the software that has been built is traceable to customer requirements

Validation: "Are we building the right product?"

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing

SOFTWARE TESTING STRATERGY – BIG PICTURE



MAJOR TYPES OF TESTING

Unit testing [white box]

- Concentrates on each component/function of the software as implemented in the source code

Integration testing

- Focuses on the design and construction of the software architecture

Validation testing

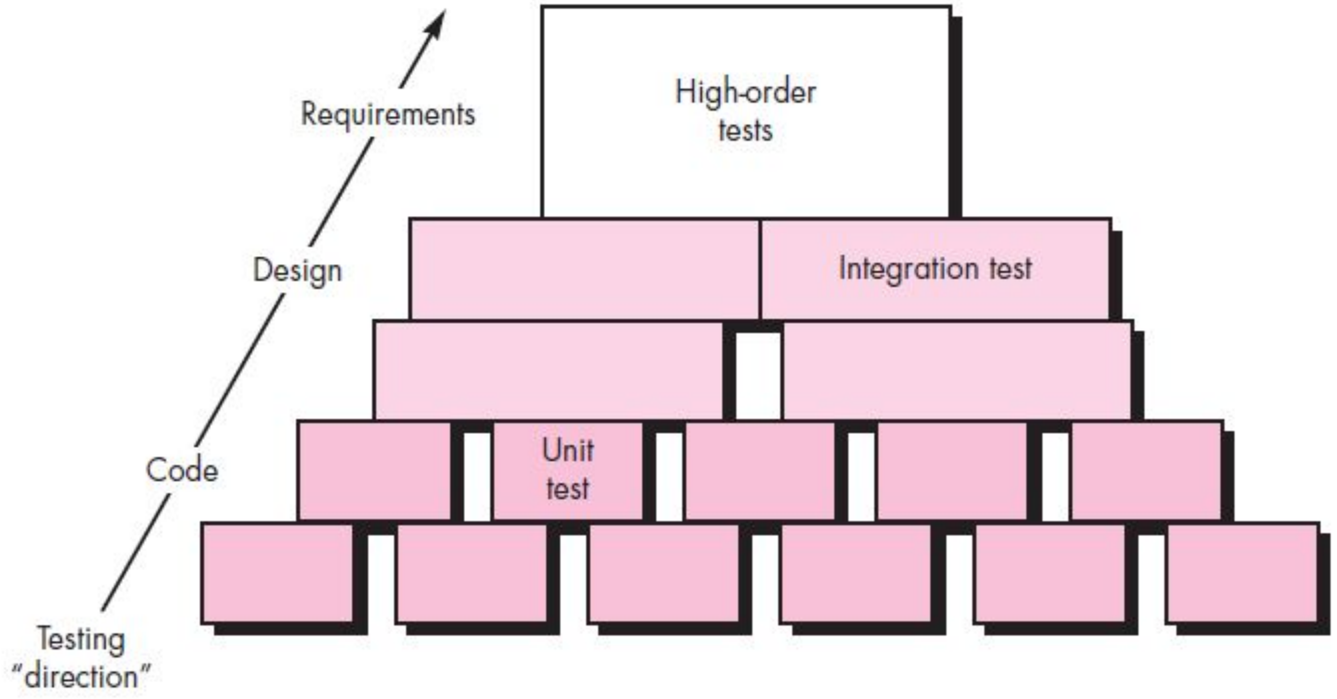
- Requirements are validated against the constructed software

System testing

- The software and other system elements are tested as a whole

- There are other sublevel testing are performed like
- Functional Testing (after Unit testing) – Testing technique used – Black box
- Regression Testing
- Smoke Testing (will see in detail in later slides)

SOFTWARE TESTING STEPS



TESTING STRATEGY APPLIED TO CONVENTIONAL APPLICATIONS

Unit testing

- Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
- Components are then assembled and integrated

Integration testing

- Focuses on inputs and outputs, and how well the components fit together and work together

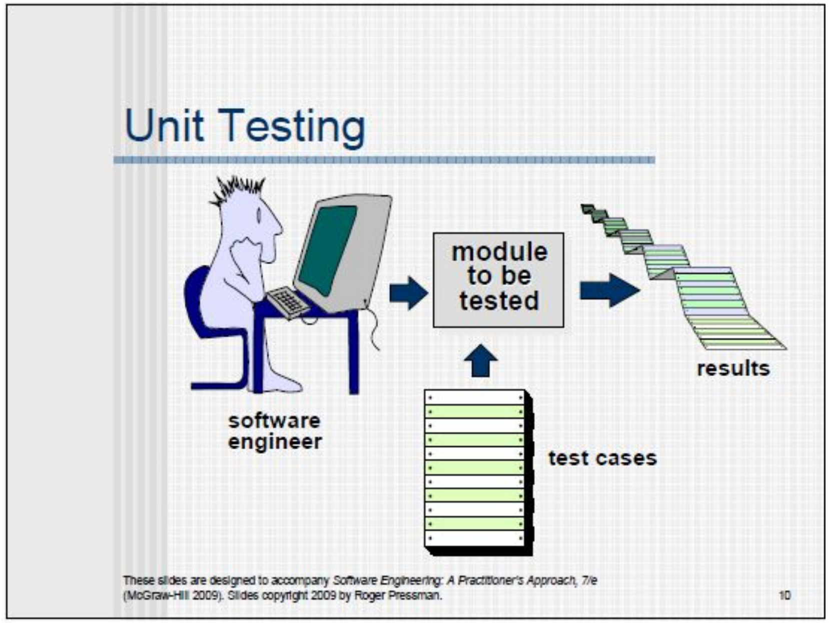
Validation testing

- Provides final assurance that the software meets all functional, behavioral, and performance requirements

System testing

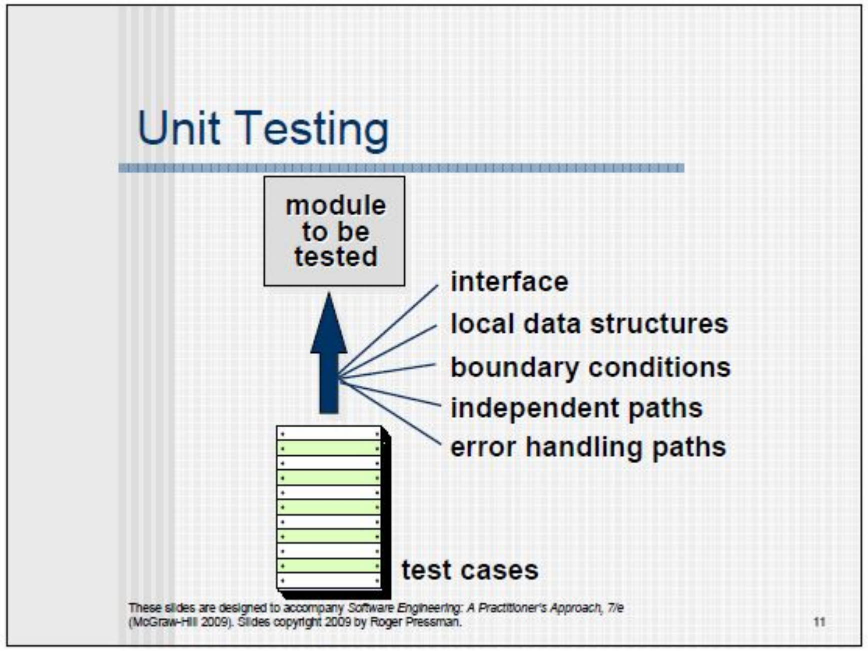
- Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

UNIT TESTING



- Individual components are tested.
- It is a path test.
- To focus on a relatively small segment of code and aim to exercise a high percentage of the internal path
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

TARGETS FOR UNIT T CASES



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Module interface
- Ensure that information flows properly into and out of the module

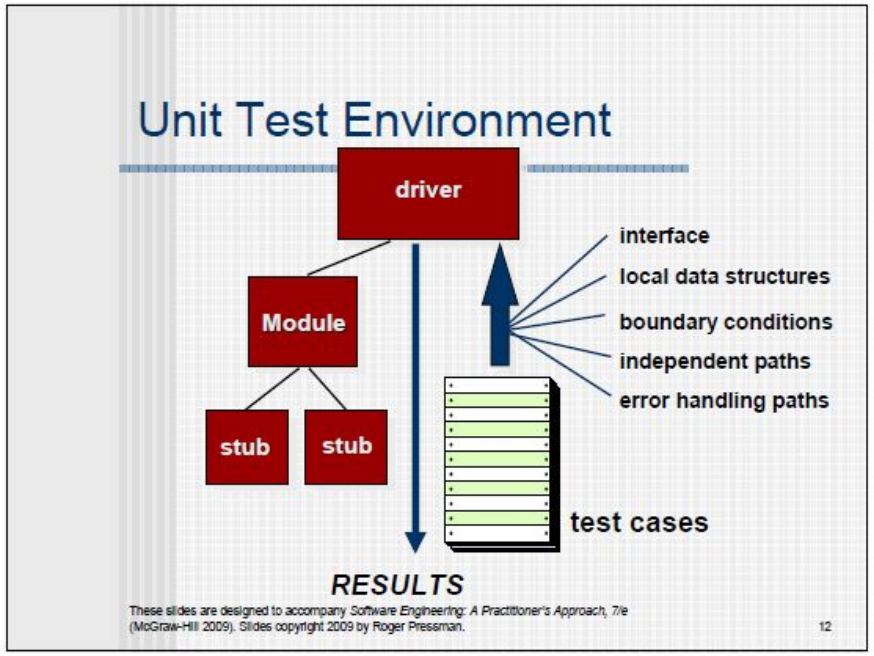
Local data structures
- Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution

Boundary conditions
- Ensure that the module operates properly at boundary values established to limit or restrict processing

Independent paths (basis paths)
- Paths are exercised to ensure that all statements in a module have been executed at least once

Error handling paths
- Ensure that the algorithms respond correctly to specific error conditions

DRIVER AND STUB FOR UNIT TESTING


Unit Test Environment
driver
Module
stub
stub
interface
local data structures
boundary conditions
independent paths
error handling paths
test cases
RESULTS
These slides are designed to accompany Software Engineering: A Practitioner's Approach, 7/e (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.
12

Driver

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

Stubs

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

Drivers and stubs both represent overhead

- Both must be written but don't constitute part of the installed software product

SOME COMMON ERROR – UNIT TESTING

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

INTEGRATION TESTING

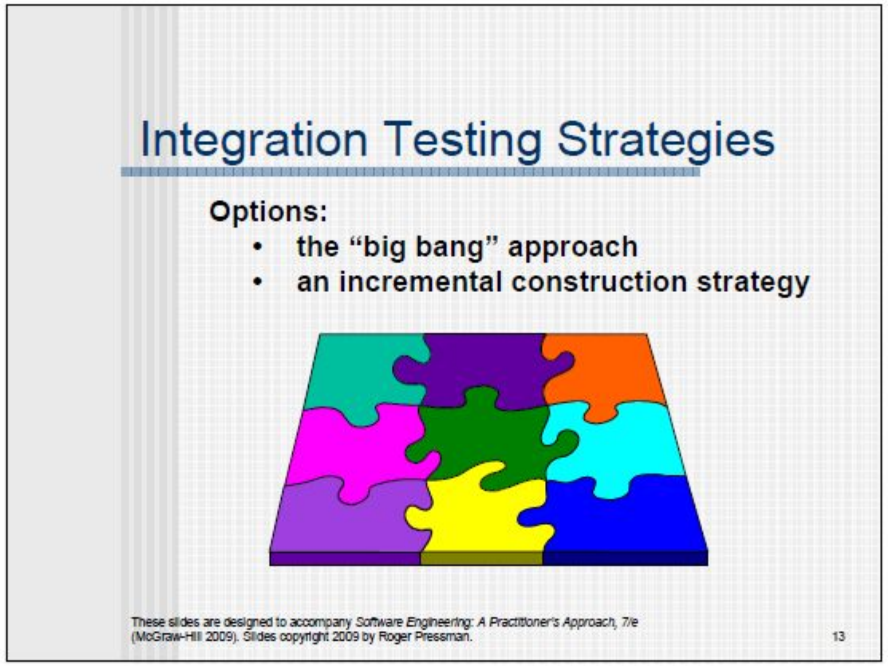
Defined as a systematic technique for constructing the software architecture

- At the same time integration is occurring, conduct tests to uncover errors associated with interfaces

Objective is to take unit tested modules and build a program structure based on the prescribed design

Two Approaches

- Non-incremental Integration Testing
- Incremental Integration Testing



Integration Testing Strategies

Options:

- the "big bang" approach
- an incremental construction strategy

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

13

NON-INCREMENTAL TESTING

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

INCREMENTAL TESTING

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

TOP-DOWN INTEGRATION TESTING

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module .
- The control program is tested first. Modules are integrated one at a time. Emphasize on interface testing
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated

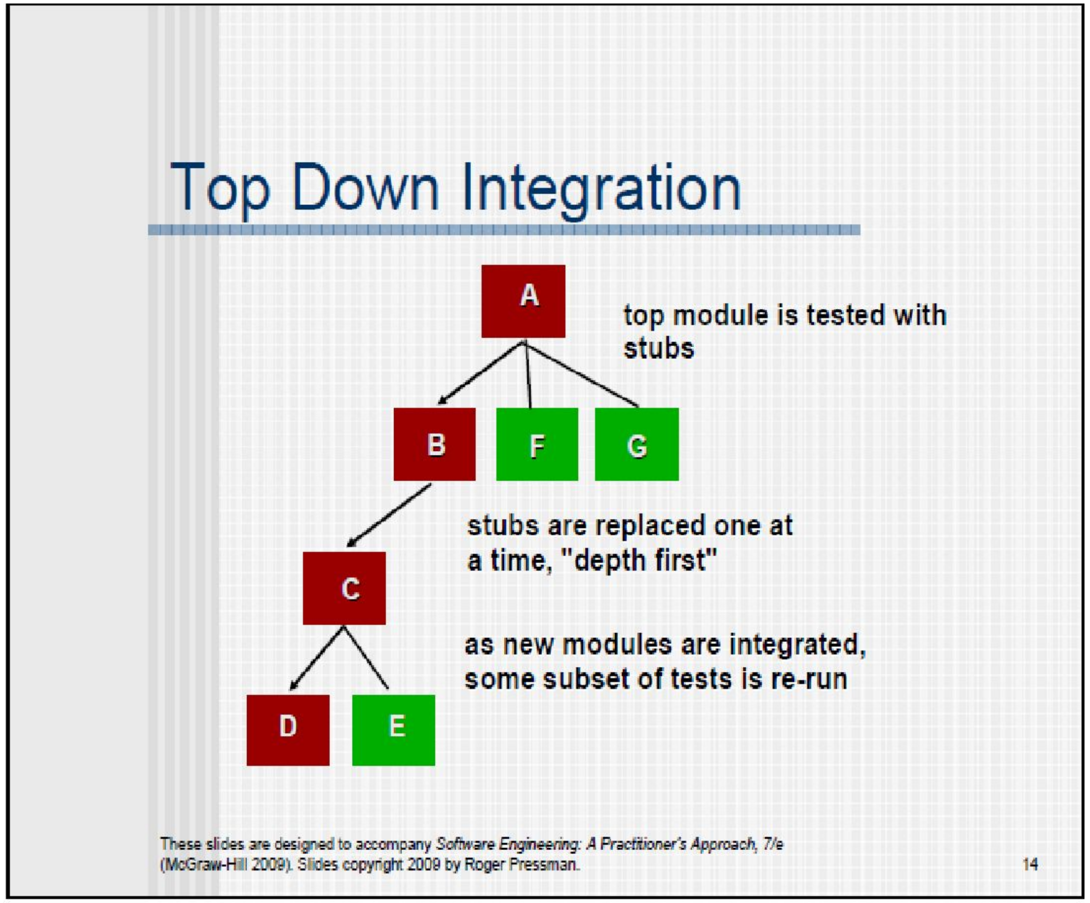
TOP DOWN INTEGARTION

Advantages

- This approach verifies major control or decision points early in the test process
- No test drivers needed
- Interface errors are discovered early
- Modular features aid debugging

Disadvantages

- Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded.
- Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process.


Top Down Integration

A
B
F
G
C
D
E
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

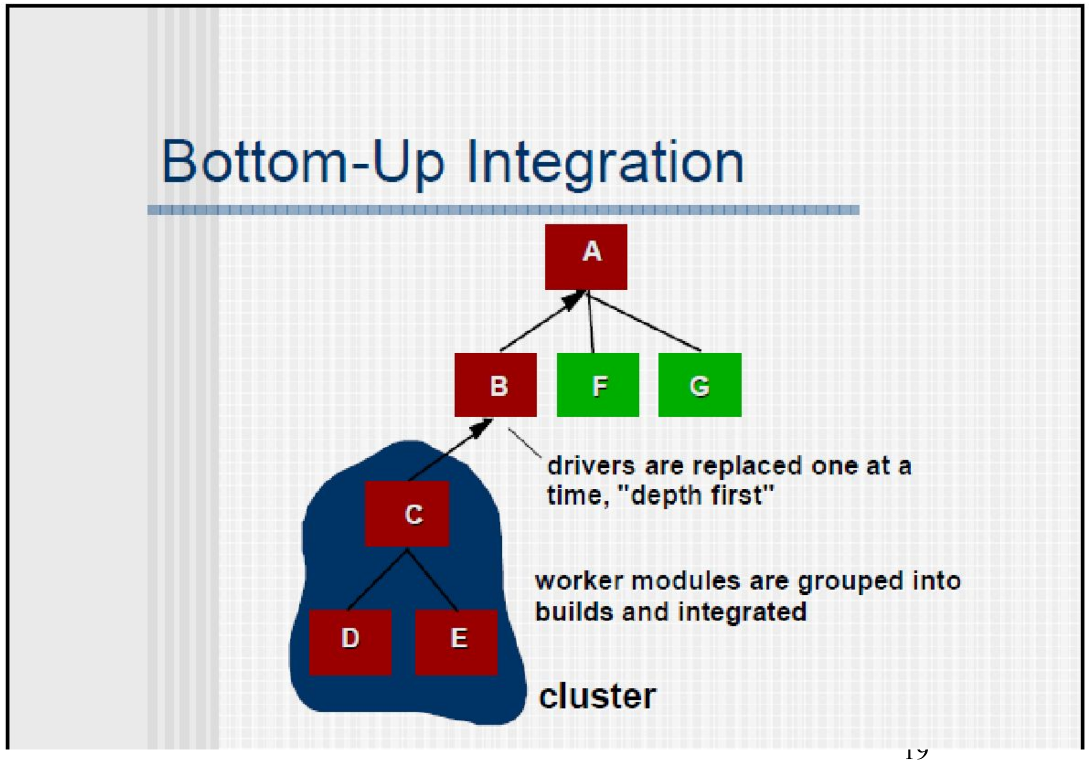
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

14

BOTTOM-UP INTEGRATION

Integration and testing starts with the most atomic modules in the control hierarchy

Allow early testing aimed at proving feasibility and emphasize on module functionality and performance


Bottom-Up Integration
A
B
F
G
C
D
E
drivers are replaced one at a time, "depth first"
worker modules are grouped into builds and integrated
cluster

BOTTOM –UP INTEGRATION

Advantages

- This approach verifies low-level data processing early in the testing process
- Need for stubs is eliminated

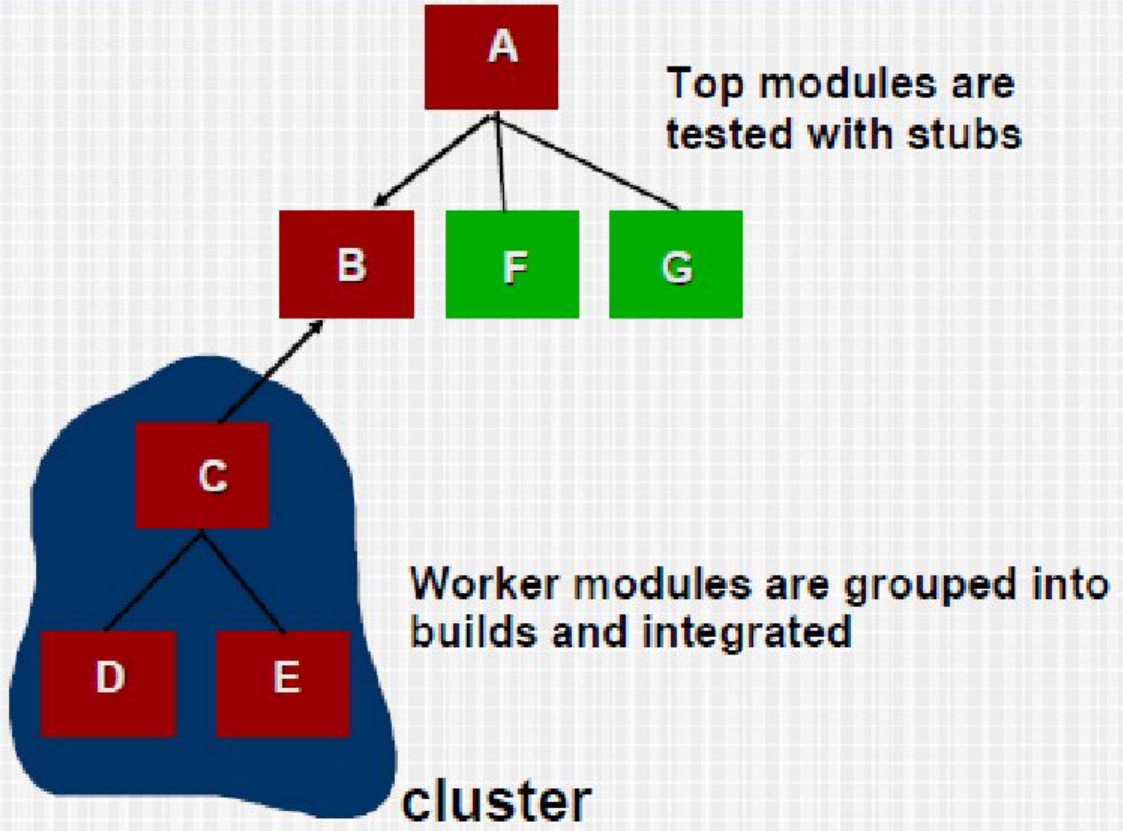
Disadvantages

- Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
- Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

SANDWICH INTEGRATION

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn’t tend towards the “big bang” scenario

Sandwich Testing


A
Top modules are tested with stubs
B
F
G
C
D
E
Worker modules are grouped into builds and integrated
cluster

REGRESSION TESTING

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed
- Regression Testing helps to ensure that changes do not introduce unintended behviour or additional errors

SMOKE TESTING

- A Common approach for creating “daily builds” for product software
- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

SMOKE TESTING - BENEFITS

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

TEST STRATEGIES – OBJECT ORIENTED SOFTWARE

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
  - Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change
  - You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class.

- You can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class
  - To illustrate the above point , consider a class hierarchy in which an operation *X* is defined for the superclass and is inherited by a number of subclasses. The context in which operation *X* is used varies in subtle ways
  - So it is necessary to test operation *X* in the context of each of the subclasses.
  - This means that testing operation *X* in a stand-alone fashion (the conventional unit-testing approach) is usually ineffective in the object-oriented context.

- class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class

- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

INTEGRATION TESTING – OO SOFTWARE

- Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies have little meaning
- There are two different strategies for integration testing of OO systems
  - 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 that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

TEST STRATEGY – WEB APPS

- Adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems
- Summary

The content model for the WebApp is reviewed to uncover errors.

2. The interface model is reviewed to ensure that all use cases can be

accommodated.

3. The design model for the WebApp is reviewed to uncover navigation errors.

4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.

5. Each functional component is unit tested.

6. Navigation throughout the architecture is tested.

7. The WebApp is implemented in a variety of different environmental configurations

and is tested for compatibility with each configuration.

8. Security tests are conducted in an attempt to exploit vulnerabilities in the

WebApp or within its environment.

9. Performance tests are conducted.

10. The WebApp is tested by a controlled and monitored population of end users.

VALIDATION TESTING

- It follows the after the integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.
- The distinction between conventional and object-oriented software disappears
- Focuses on user-visible actions and user-recognizable output from the system
- Demonstrates conformity with requirements
- Designed to ensure that
   - All functional requirements are satisfied
   - All behavioral characteristics are achieved

CONTD..

- All performance requirements are attained
- Documentation is correct
- Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

After each validation test

- The function or performance characteristic conforms to specification and is accepted
- A deviation from specification is uncovered and a deficiency list is created

A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

ALPHA AND BETA TESTING

Alpha testing

- Conducted at the developer's site by end users
- Software is used in a natural setting with developers watching intently
- Testing is conducted in a controlled environment

Beta testing

- Conducted at end-user sites
- Developer is generally not present
- It serves as a live application of the software in an environment that cannot be controlled by the developer
- The end-user records all problems that are encountered and reports these to the developers at regular intervals

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

ACCEPTANCE TESTING

- A variation on beta testing, called *customer acceptance testing,* is sometimes performed when custom software is delivered to a customer under contract.
- The customer performs a series of specific tests in an attempt to uncover errors beforeaccepting the software from the developer.
- In some cases (e.g., a major corporate orgovernmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SYSTEM TESTING

- Recovery Testing
- Security Testing
- Stress Testing
- Performance Testing
- Deployment Testing

RECOVERY TESTING & SECURITY TESTING

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access or penetration

STRESS TESTING

- Stress testing

  Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

  Examples

  (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate,

  (2) input data rates may be increased by an order of magnitude to determine how input functions will respond,

  (3) test cases that require maximum memory or other resources are executed,

  (4) test cases that may cause thrashing in a virtual operating system are designed,

  (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

SENSITIVITY TESTING

- Variation of Stress Testing
- In some situations (the most common occur in mathematical algorithms), a very small range ofdata contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

PERFORMANCE TESTING

- Tests the run-time performance of software within the context of an integrated system
- Often coupled with stress testing and usually requires both hardware and software instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis.
- Can uncover situations that lead to degradation and possible system failure

DEPLOYMENT TESTING

- *Deployment testing,* sometimes called *configuration testing,* exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.
- A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows).
- Because security is a major issue, a complete set of security tests would be integrated with the deployment test.

TESTING FUNDAMENTALS - CHARACTERISTICS

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently
- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

TEST CHARACTERISTICS

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be "best of breed"
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

TWO UNIT TESTING TECHNIQUES

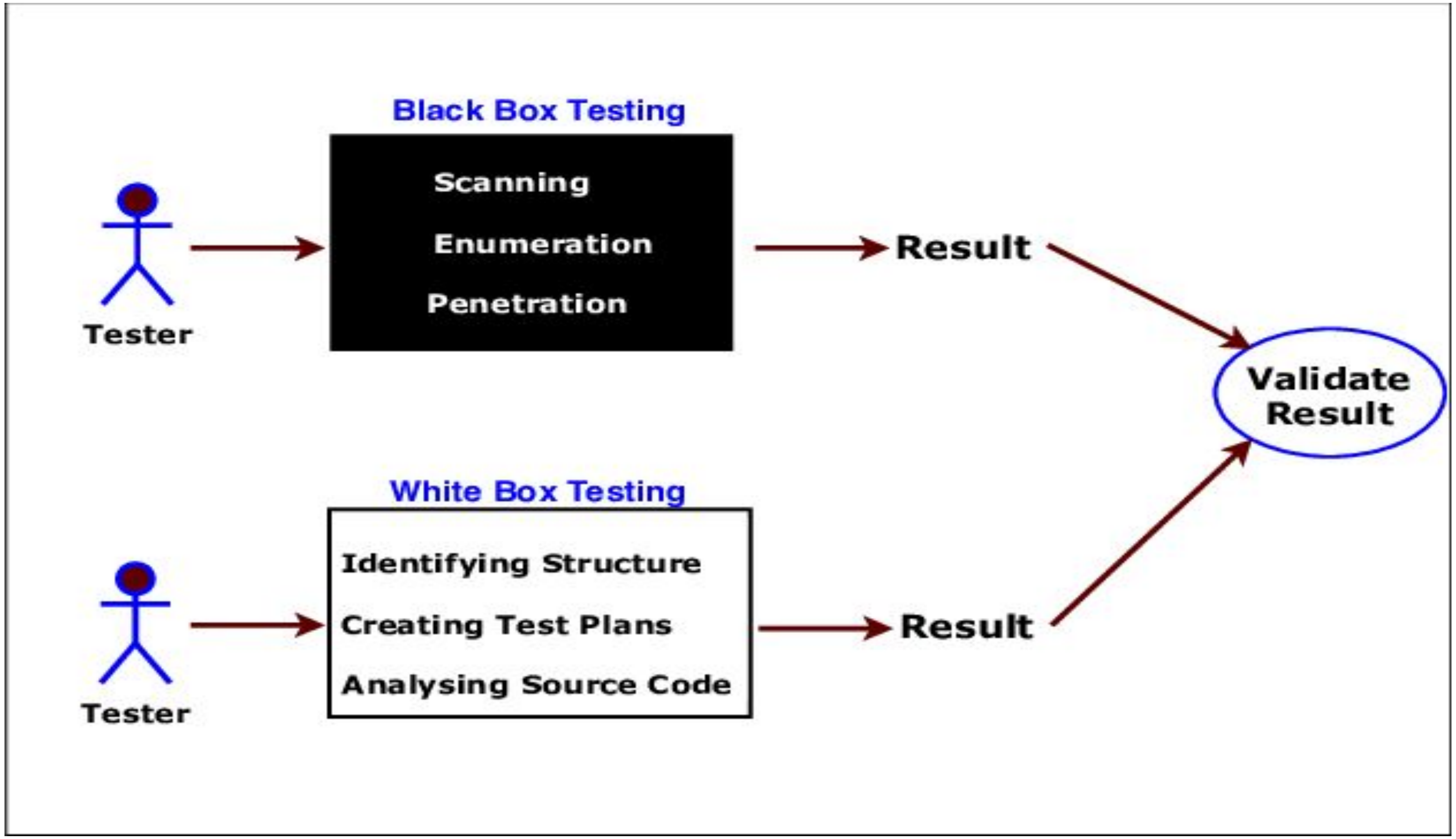
Black-box testing

- Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
- Includes tests that are conducted at the software interface
- Not concerned with internal logical structure of the software

White-box testing

- Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
- Involves tests that concentrate on close examination of procedural detail
- Logical paths through the software are tested
- Test cases exercise specific sets of conditions and loops

BLACK BOX VS WHITE BOX



WHITE BOX TESTING

- *White-box testing,* sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases
- White box testing guarantees
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

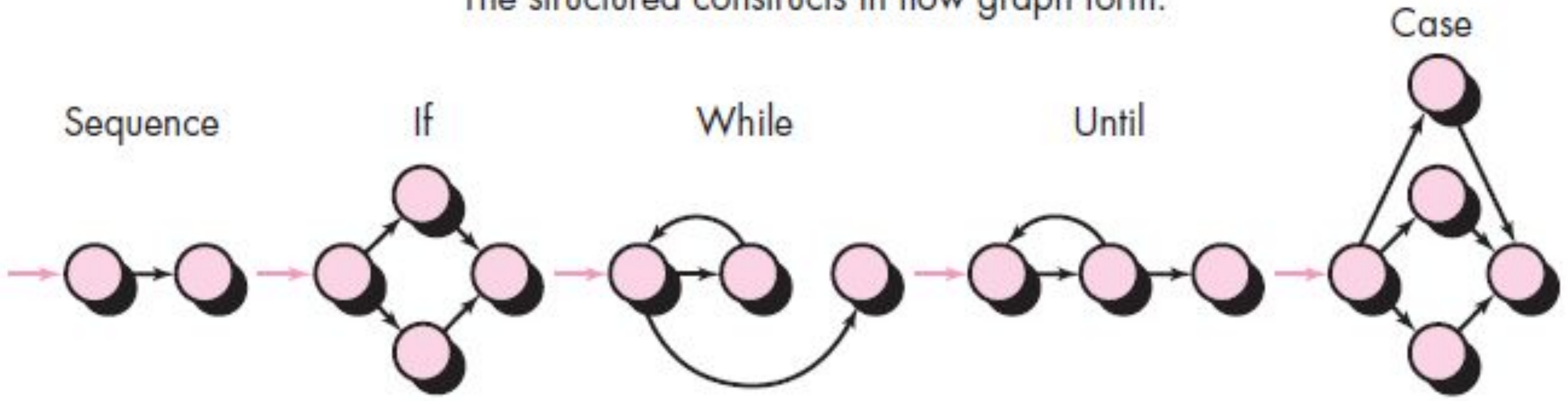
BASIS PATH TESTING

- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute <u>every statement</u> in the program <u>at least one time</u> during testing

FLOW GRAPH NOTATION

- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge

The structured constructs in flow graph form:

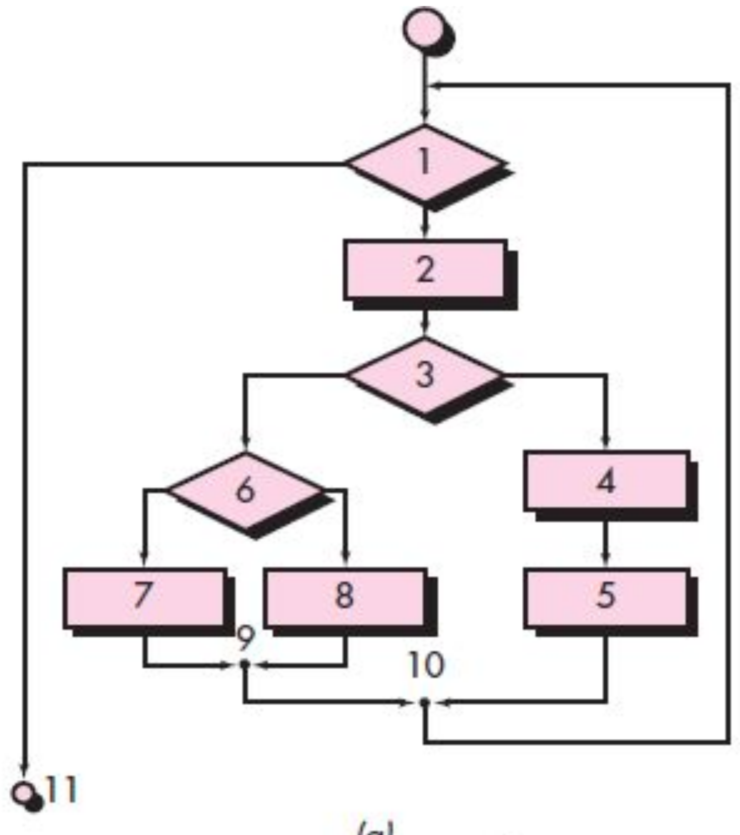


Where each circle represents one or more nonbranching PDL or source code statements

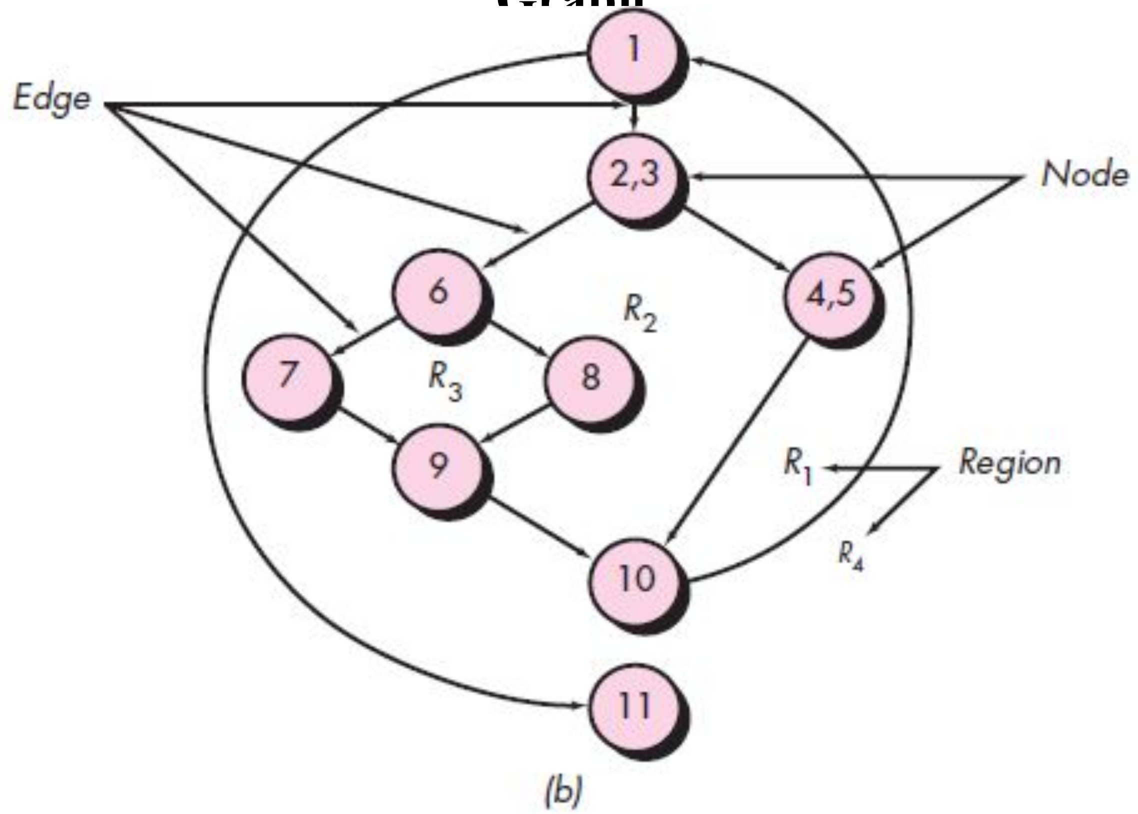
FLOW GRAPH EXAMPLE

- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Flow Chart



Flow Graph



INDEPENDENT PROGRAM PATHS

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
- Path 1: 0-1-11
- Path 2: 0-1-2-3-4-5-10-1-11
- Path 3: 0-1-2-3-6-8-9-10-1-11
- Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity
- Paths 1 through 4 constitute a *basis set* for the flow graph

CYCLOMATIC COMPLEXITY

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions
  - V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
- Number of regions = 4
- V(G) = 11 edges – 9 nodes + 2 = 4
- V(G) = 3 predicate nodes + 1 = 4

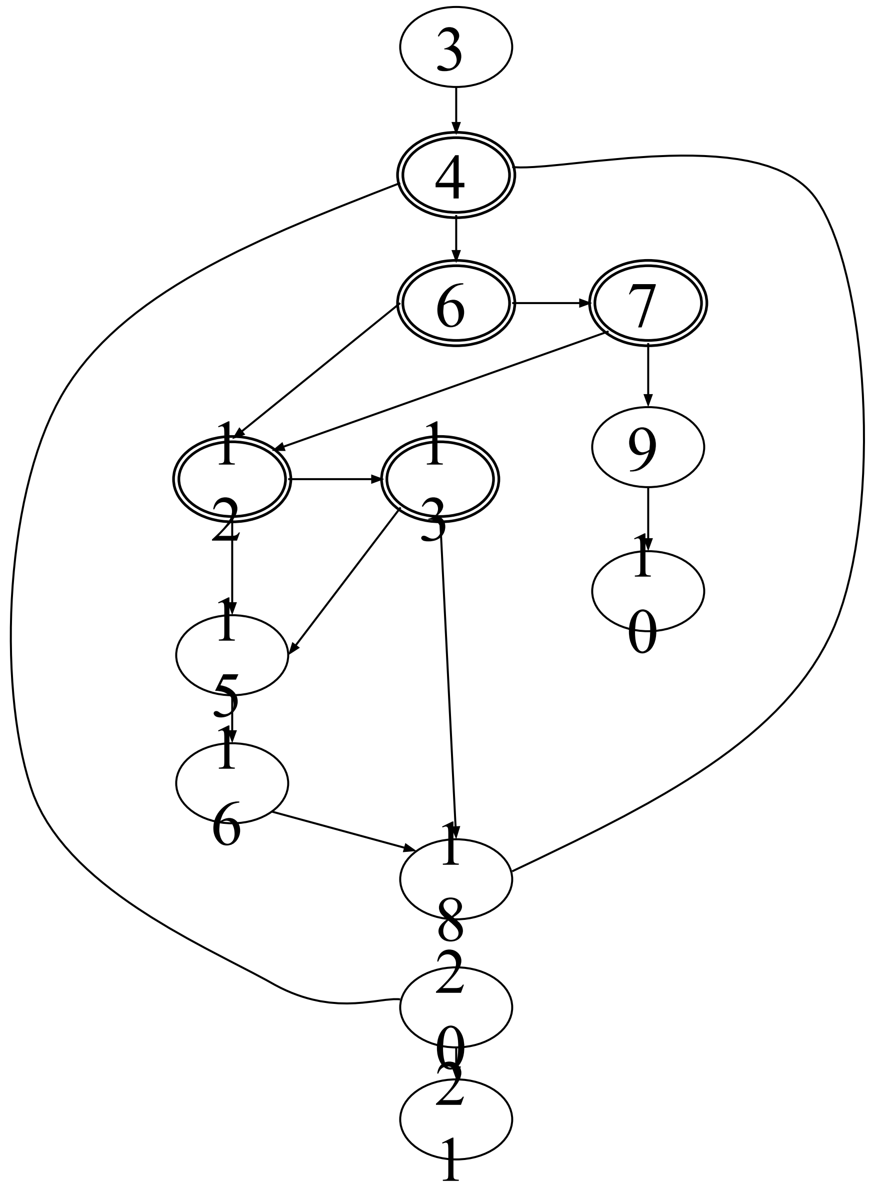
Therefore cyclomatic complexity = 4

EXAMPLE

```
 1   int functionZ(int y)
 2   {
 3   int x = 0;

 4   while (x <= (y * y))
 5       {
 6       if ((x % 11 == 0) &&
 7            (x % y == 0))
 8           {
 9           printf(“%d”, x);
10           x++;
11           } // End if
12       else if ((x % 7 == 0) ||
13                 (x % y == 1))
14           {
15           printf(“%d”, y);
16           x = x + 2;
17           } // End else
18       printf(“\n”);
19       } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```

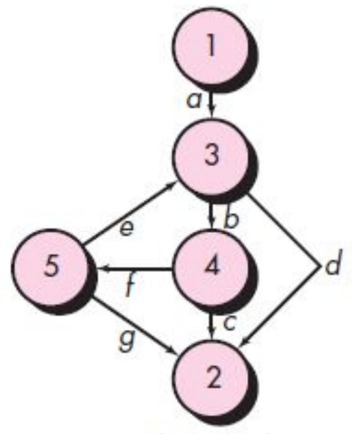


GRAPH MATRICES

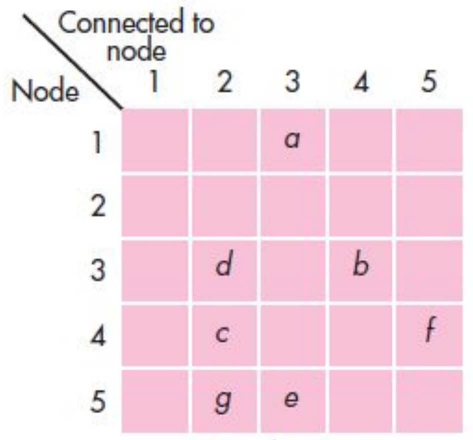
- The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization.
- A data structure, called a *graph matrix,* can be quite useful for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

GRAPH MATRICES

- Each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*.



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

- The link weight provides additional information about control flow

GRAPH MATRICES

In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- • The probability that a link (edge) will be execute.
- • The processing time expended during traversal of a link
- • The memory required during traversal of a link
- • The resources required during traversal of a link.

Using these techniques, the analysis required to design test cases can be partially or fully automated.

DERIVING THE BASIS SET AND TEST CASES

1) Using the design or code as a foundation, draw a corresponding flow graph
2) Determine the cyclomatic complexity of the resultant flow graph
3) Determine a basis set of linearly independent paths
4) Prepare test cases that will force execution of each path in the basis set

CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. Hence there is a variation on control structure testing which broadens the testing coverage and improve the quality of white-box testing.

Condition Testing

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

- E1 <relational-operator> E2

where E1 and E2 are arithmetic expressions and <relational-operator> is one of the

following: < , <=,=, not equal, > , >=

CONDITION TESTING

- A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (), AND (&),and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.
- If a condition is incorrect, then at least one component of the condition is incorrect.
- The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

LOOP TESTING

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests

A white-box testing technique that focuses exclusively on the validity of loop constructs

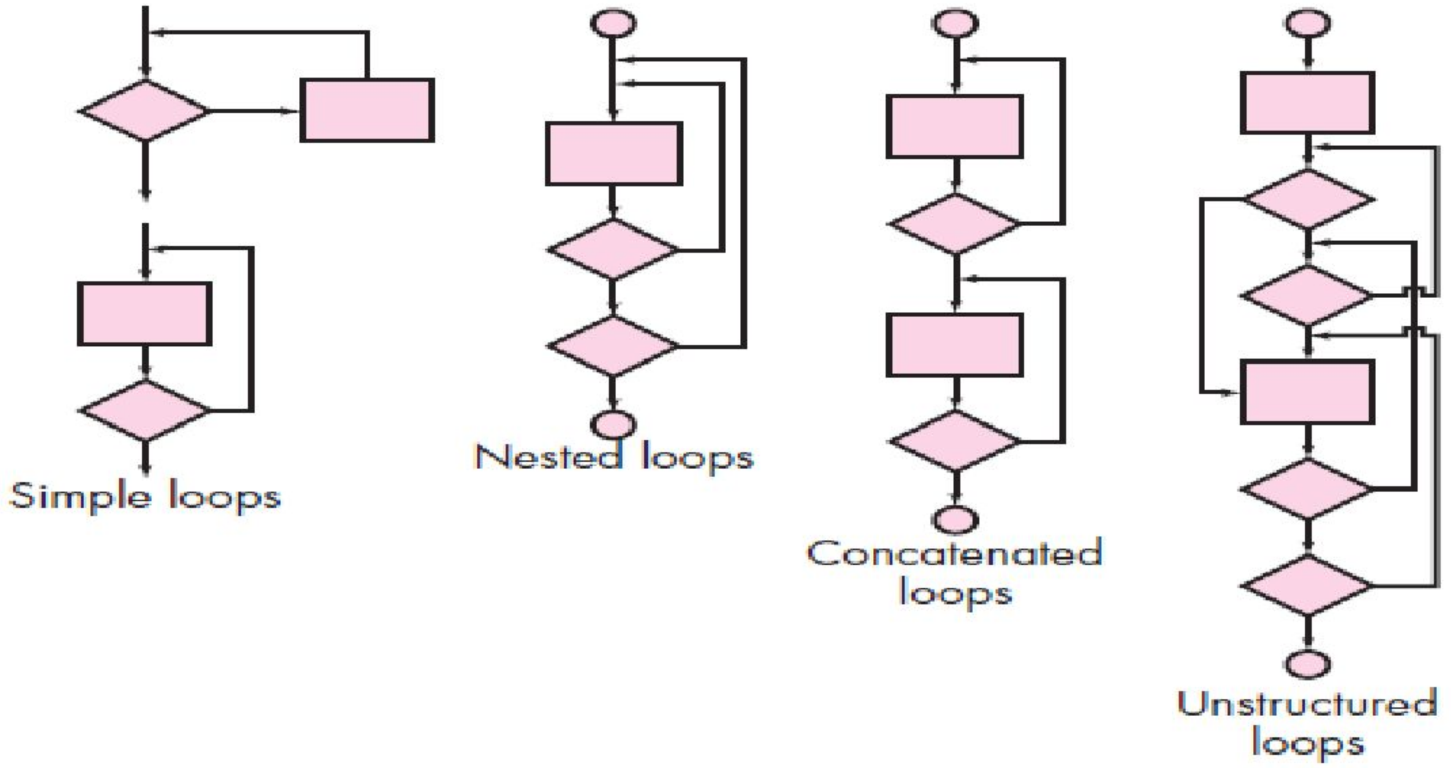
Four different classes of loops exist
- Simple loops
- Nested loops
- Concatenated loops
- Unstructured loops

Testing occurs by varying the loop boundary values
- Examples:

```
for (i = 0; i < MAX_INDEX; i++)

while (currentTemp >= MINIMUM_TEMPERATURE)
```

CLASSES OF LOOPS



TESTING OF 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

'n' is the maximum number of allowable passes through the loop

TESTING OF 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 values; add other tests for out-of-range or excluded values
3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values
4) Continue until all loops have been tested

TESTING OF CONCATENATED LOOPS

- Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
- However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
- When the loops are not independent,the approach applied to nested loops is recommended

TESING OF UNSTRUCTURED LOOPS

Redesign the code to reflect the use of structured programming practices

Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

BLACK BOX TESTING

Black-box testing, also called *behavioural testing,* focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than whitebox methods.

Black-box 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) behaviour or performance errors, and

(5) initialization and termination errors.

QUESTIONS ANSWERED BY BLACK BOX

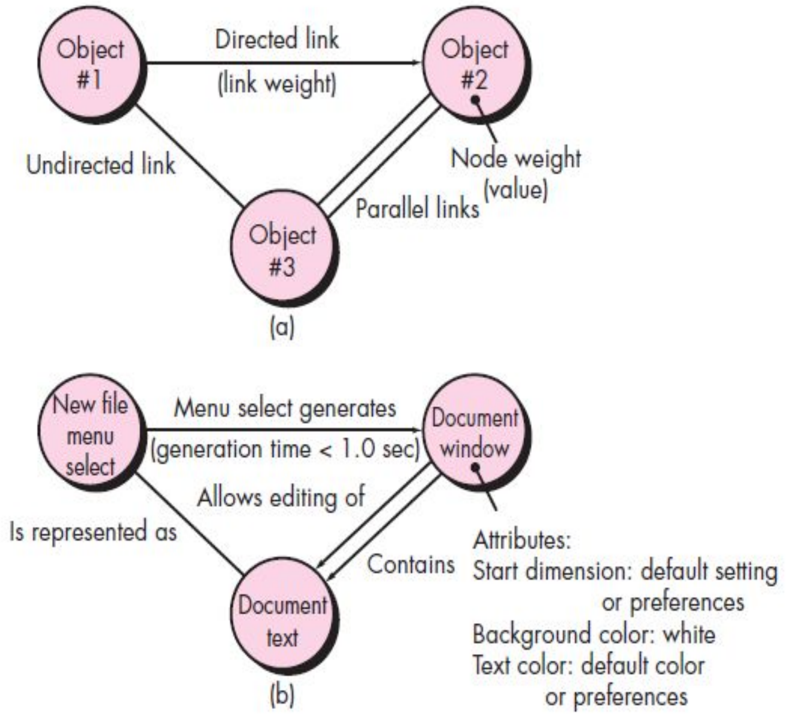
1. How is functional validity tested?
2. How are system behavior and performance tested?
3. What classes of input will make good test cases?
4. Is the system particularly sensitive to certain input values?
5. How are the boundary values of a data class isolated?
6. What data rates and data volume can the system tolerate?
7. What effect will specific combinations of data have on system operation?

GRAPH BASED TESTING METHOD

- The first step in black-box testing is to understand the objects5 that are modelled in software and the relationships that connect these objects
- the next step is to define a series of tests that verify "all objects have the expected relationship to one another"
- Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

STEPS

1. creating a *graph*—a collection of *nodes* that represent objects,
2. *links* that represent the relationships between objects,
3. *Node weights* that describe the properties of a node
4. *link weights* that describe some characteristic of a link



1. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction.
2. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions.
3. *Parallel links* are used when a number of different relationships are established between graph nodes

After creating the graph derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designedin an attempt to find errors in any of the relationships.

BEHAVIORAL TESTING METHODS

1. **Transaction flow modeling.** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps

2. **Finite state modeling.** The nodes represent different user-observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state

3. **Data flow modeling.** The nodes are data objects, and the links are the transformations that occur to translate one data object into another.

4. **Timing modeling.** The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

EQUIVALENCE PARTITIONING

A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived

An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed

Test case design is based on an evaluation of equivalence classes for an input condition

An equivalence class represents a set of valid or invalid states for input conditions

Typically, an inputcondition is either a specific numeric value, a range of values, a set of related values,or a Boolean condition.

From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

CONTD

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the later stages of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand

GUIDELINES FOR DEFINING EQUIVALENCE CLASSES

If an input condition specifies a range, one valid and two invalid equivalence classes are defined

- Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}

If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

- Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}

If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined

- Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}

If an input condition is a Boolean value, one valid and one invalid class are define

- Input: {true condition} Eq classes: {true condition}, {false condition}

BOUNDARY VALUE ANALYSIS

A greater number of errors occur at the boundaries of the input domain rather than in the "center"

Boundary value analysis is a test case design method that complements equivalence partitioning

- It selects test cases at the edges of a class
- It derives test cases from both the input domain and output domain

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the “edges” of the class

GUIDELINES FOR BOUNDARY VALUE ANALYSIS

1. If an input condition specifies a <u>range</u> bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*

2. If an input condition specifies a <u>number of values</u>, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested

3. Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above

4. If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

ORTHOGONAL ARRAY TESTING

In certain application the number of input parameters is small and the values that each of the parameters may take are clearly bounded.

When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain

However, as the number of input values grows and the number of discrete values for each data item increases,exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing

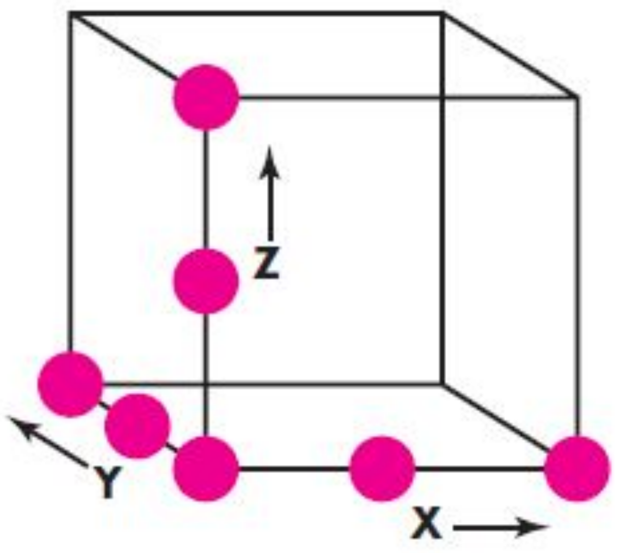
ORTHOGONAL ARRAY TESTING

The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

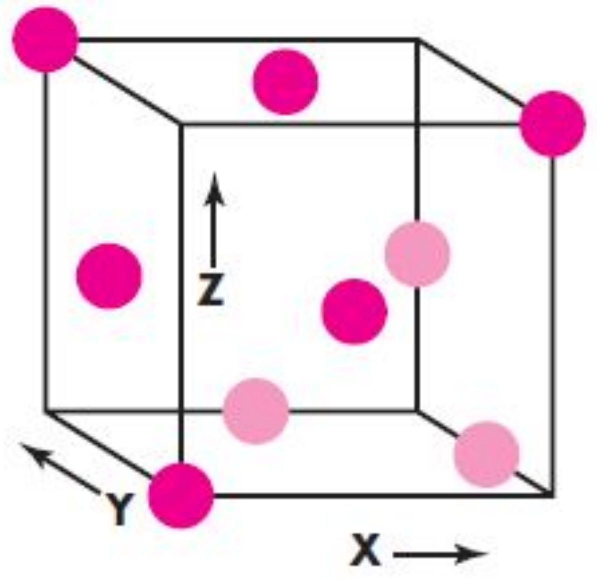
one input item at a time” approaches, consider a system that has three input items, *X, Y,* and *Z.* Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases

A geometric view of the possible test cases associated with X, Y, and Z

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property". That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube. Test coverage across the input domain is more complete.



One input item at a time



L9 orthogonal array

CONTD

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is $3^{4} = 81$, large but manageable.

All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy.

MODEL BASED TESTING

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases.

In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model, as the basis for the design of test cases

STEPS OF MODEL BASED TESTING

1. Analyze an existing behavioral model for the software or create one

2. Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state

3. Review the behavioral model and note the expected outputs as the software makes the transition from state to state

4. Execute the test cases

5. Compare actual and expected results and take corrective action as required

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

TOPICS

Planning – Testing

Test Prioritization

Effort Estimation

Test Point Analysis

TEST STRATEGY AND PLANNING

- There are many techniques available to execute software test projects. It depends on the kind of test project.
- Most test projects must have a test plan and a test strategy before the project can be ready for execution.
- Due to time constraints, testing cycles are cut shorted. This leads to a half-tested product that is pushed out the door.

TEST PLANNING

Test planning is an essential practice for any organization that wishes to develop a test process that is repeatable and manageable

Test planning should begin early in the software life cycle, although for many prganizations whose test processes are immature this practice is not yet in place

TEST PLANNER - STEPS

Test plans for software projects are very complex and detailed documents.

The planner usually includes the following essential high-level items.

1. *Overall test objectives.* As testers, why are we testing, what is to be achieved by the tests, and what are the risks associated with testing this product?

2. *What to test (scope of the tests).* What items, features, procedures, functions, objects, clusters, and subsystems will be tested?

3. *Who will test.* Who are the personnel responsible for the tests?

4. *How to test.* What strategies, methods, hardware, software tools, and techniques are going to be applied? What test documents and deliverable should be produced?

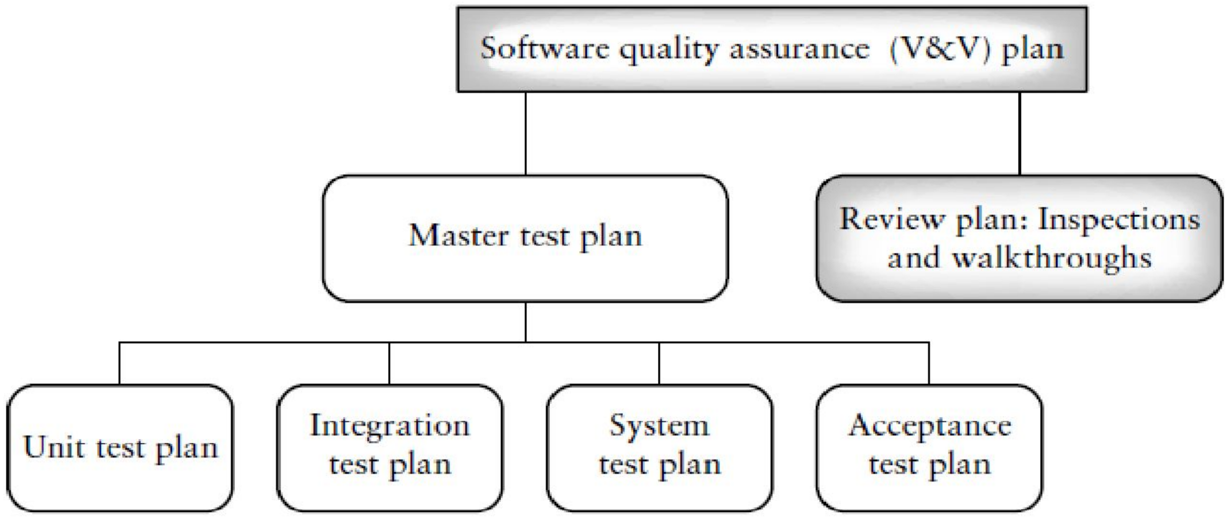
5. *When to test.* What are the schedules for tests? What items need to be available?

6. *When to stop testing.* It is not economically feasible or practical toplan to test until all defects have been revealed. This is a goal that testers can never be sure they have reached.

TEST PLAN - HIERARCHY

- Test plans can be organized in several ways depending on organizational policy.
- There is often a hierarchy of plans that includes several levels of quality assurance and test plans.
- The complexity of the hierarchy depends on the type, size, risk-proneness, and the mission/safety criticality of software system being developed.
- Depending on organizational policy, another level of the hierarchy could contain a separate test plan for unit, integration, system,and acceptance tests.
- The level-based plans give a more detailed view of testing appropriate to that level.
- The persons responsible for developing test plans depend on the type of plan under development.
- Usually staff from one or more groups cooperates in test plan development.

- At the top of the plan hierarchy there may be a software quality assurance plan.
- This plan gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, configuration control, and supplier control.
- The in the plan hierarchy there may be a master test plan that includes an overall description of all execution-based testing for the software system.
- A master verification plan for reviews inspections/walkthroughs would also fit in at this level.
- The master test plan itself may be a component of the overall project plan or exist as a separate document.



Testing Strategies

Identifying the mistakes by the end-user is costly. Fixing these defects at this stage is costly.

Test strategies should include things like test prioritization, automation strategy, risk analysis, etc.

Test planning should include a work breakdown structure, requirement review, resource allocation, effort estimation, tools selection, setting up communication channels, etc.

TEST PRIORITIZATION

- All parts of the software product will not be used by end users with the same intensity. Some parts of the product are used by end users extensively, while other parts are seldom used.
- So the extensively used parts of the product should not have any defects at all and thus they need to be tested thoroughly.
- High priority on tests which are to be done for these critical parts of the software product and put a low priority on uncritical parts.
- The high priority areas are first tested.
- Once testing is thoroughly done for these parts, then you should start testing low priority areas.

EFFORT ESTIMATION

- Effort estimate should include information such as project size, productivity, and test strategy.
- **wideband Delphi technique** uses brainstorming sessions to arrive at effort estimate figures after discussing the project details with the project team.
- This is a good technique because the people who will be assigned the project work will know their own productivity levels and can figure out the size of their assigned project tasks from their own experience.
- Experience-based technique, instead of group sessions, the test manager meets each team member and asks his estimate for the project work he has been assigned.

TEST POINT ANALYSIS

Testing process needs the estimation of efforts, required in the terms of budget, resources and time.

It helps a QA team to plan the whole testing phase with respect to the estimated efforts and to execute the testing process effectively and efficiently within the stipulated time period.

There are multiple number of methods available to estimate the testing efforts.

- One such method is the Test Point Analysis

TEST POINT ANALYSIS

- Test Point Analysis (TPA) is a software testing estimate approach, which is specifically designed to estimate the black box testing efforts.
- Unlike, the Functional point Analysis (FPA), which is used to project the White box testing efforts, TPA is used to estimate the black box testing efforts, especially in performing the system and the acceptance test of the software application.
- The test point analysis technique consists of three inputs to estimate the black box testing efforts : project size, Test strategy, and productivity.
- Test points, in turn, are calculated from function points. The number of function points is calculated from the number of functions and function complexity.

TAP calculates the test efforts estimation in test points for highly important functions, according to user and for whole system.

As the test points of functions are measured, the tests can test important functionalities first then be predicting the risk in testing.

This technique also consider the quality characteristics, such as functionality, security, usability, efficiency, etc. with proper weighting process for calculating TAP

Static testing: It's a verification activity which examines the whole structure of the application without executing it.

Dynamic testing: It's an actual testing activity performed over the application to evaluate and assess its different features, functionalities and quality aspects. It may consist of both explicit and implicit type of testing.

Test Point Analysis - Steps

i) a) Dynamic and static test points are calculated. Dynamic test points are number of test which are based on dynamic measurable quality. Characteristic of functions in system. To calculate dynamic test point.

- Function points (FPs) assigned to function.
- Function dependent factor (FDC) such as interfacing, function importance, etc.
- Quality characteristics (QC)

b) Static test points are number of test points which are based on quality characteristics of system, it is calculated based on –

- Function points (FPs) assigned to system.
- Quality requirements or test strategy for static quality requirements (QC) like Flexibility, Testability, security,continuity,traceability.

ii) Dynamic test point is added to static test point. To get total test points (TTP) for system.

iii) Total Test Points is used for calculation of primary test hours (PTH) PTH is effort estimation for primary test hour’s activity such as preparation for primary test hour’s activity such as preparation, specification and execution.

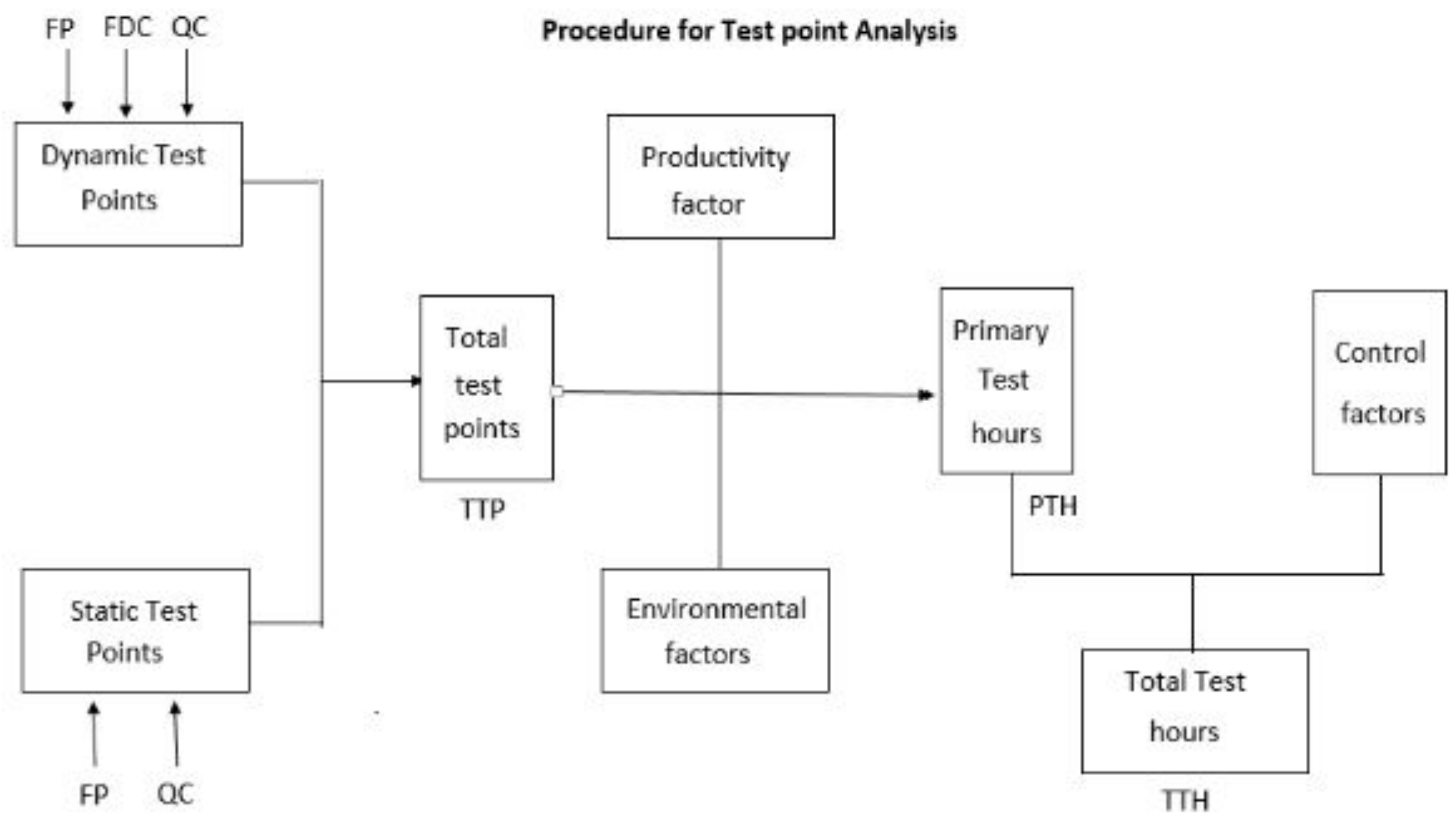
iv) PTH is calculated based on environmental factors and productivity factors.

v) Secondary testing activities include management activities like controlling the testing activities.

vi) TTH is calculated by adding some allowance to secondary activities and PTH.

vii) Thus TTH is final effort estimation for testing activities.

TEST POINT ANALYSIS



TEST PROJECT MONITORING AND CONTROL

Project monitoring (or tracking) refers to the activities and tasks managers engage in to periodically check the status of each project. Reports are prepared that compare the actual work done to the work that was planned.

Monitoring requires a set of tools, forms, techniques, and measures. A precondition for monitoring a project is the existence of a project plan.

PROJECT CONTROLLING

- Project controlling consists of developing and applying a set of corrective actions to get a project on track when monitoring shows a deviation from what was planned.
- If monitoring results show deviations from the plan have occurred, controlling mechanisms must be put into place to direct the project back on its proper track.
- Controlling a project is an important activity which is done to ensure that the project goals will be achieved occurring to the plan

MAJOR TASKS – PROJECT CONTROLLING

1. ***Develop standards of performance*** - These set the stage for defining goals that will be achieved when project tasks are correctly accomplished

2. ***Plan each project***. - The plan must contain measurable goals,milestones,deliverables, and well-defined budgets and schedules that take into consideration project types, conditions, and constraints.

3. ***Establish a monitoring and reporting system*** - In the monitoring and reporting system description the organization must describe the measures to be used, how/when they will be collected, what questions they will answer, who will receive the measurement reports, and how these will be used to control the project. Each project plan must describe the monitoring and reporting mechanisms

4. ***Measure and analyze results. -*** Measurements for monitoring and controlling must be collected, organized, and analyzed. They are then used to compare the actual achievements with standards, goals, and plans.

5. ***Initiate corrective actions for projects that are off track***. These actions may require changes in the project requirements and the project plan.

6. ***Reward and discipline***. Reward those staff who have shown themselves to be good performers, and discipline, retrain, relocate those that have consistently performed poorly.

7. ***Document the monitoring and controlling mechanisms***. All the methods,forms, measures, and tools that are used in the monitoring and controlling process must be documented in organization standards and be described in policy statements.

8. ***Utilize a configuration management system***. A configuration management system is needed to manage versions, releases, and revisions of documents, code, plans, and reports.

Test Case Design

Master Test Plan

Test Case Management

- Test Bed Preparation
- Test Case Execution
- Defect Tracking

Test Reporting

Test Artifacts

TEST CASE DESIGN

A good test case design technique is crucial to improving the quality of the software testing process. This helps to improve the overall quality and effectiveness of the released software.

The main purpose of test case design techniques is to test the functionalities and features of the software with the help of effective test cases.

The test case design techniques are broadly classified into three major categories.

1. Specification-Based techniques
2. Structure-Based techniques
3. Experience-Based techniques

SPECIFICATION – BASED TESTING

1. Specification-Based or Black-Box techniques

This technique leverages the external description of the software such as technical specifications, design, and client’s requirements to design test cases. The technique enables testers to develop test cases that provide full test coverage. The Specification-based or black box test case design techniques are divided further into 5 categories. These categories are as follows:

1. **Boundary Value Analysis (BVA)**

2. Equivalence Partitioning (EP)

3. Decision Table Testing

4 State Transition Diagrams

5. Use Case Testing

1. Boundary Value Analysis (BVA)

This technique is applied to explore errors at the boundary of the input domain. BVA catches any input errors that might interrupt with the proper functioning of the program.

2. Equivalence Partitioning (EP)

In Equivalence Partitioning, the test input data is partitioned into a number of classes having an equivalent number of data. The test cases are then designed for each class or partition. This helps to reduce the number of test cases.

3. Decision Testing Coverage

This technique is also known as branch coverage is a testing method in which each one of the possible branches from each decision point is executed at least once to ensure all reachable code is executed. This helps to validate all the branches in the code. This helps to ensure that no branch leads to unexpected behavior of the application.

4. State Transition Diagrams

In this technique, the software under test is perceived as a system having a finite number of states of different types. The transition from one state to another is guided by a set of rules. The rules define the response to different inputs. This technique can be implemented on the systems which have certain workflows within them.

5. Use Case Testing

A use case is a description of a particular use of the software by a user. In this technique, the test cases are designed to execute different business scenarios and end-user functionalities. Use case testing helps to identify test cases that cover the entire system.

STRUCTURE BASED / WHITE-BOX TECHNIQUE

The structure-based or white-box technique design test cases based on the internal structure of the software.

This technique exhaustively tests the developed code. Developers who have complete information of the software code, its internal structure, and design help to design the test cases.

This technique is further divided into five categories.

1. **Statement Testing & Coverage**

2. **Decision Testing Coverage**

3. **Condition Testing**

4. Multiple Condition Testing

5. All Path Testing

Statement Testing & Coverage

This technique involves execution of all the executable statements in the source code at least once. The percentage of the executable statements is calculated as per the given requirement. This is the least preferred metric for checking test coverage.

Decision Testing Coverage

This technique is also known as branch coverage is a testing method in which each one of the possible branches from each decision point is executed at least once to ensure all reachable code is executed. This helps to validate all the branches in the code. This helps to ensure that no branch leads to unexpected behavior of the application.

Condition Testing

Condition testing also is known as Predicate coverage testing, each Boolean expression is predicted as TRUE or FALSE. All the testing outcomes are at least tested once. This type of testing involves 100% coverage of the code. The test cases are designed as such that the condition outcomes are easily executed.

Multiple Condition Testing

The purpose of Multiple condition testing is to test the different combination of conditions to get 100% coverage. To ensure complete coverage, two or more test scripts are required which requires more efforts.

All Path Testing

In this technique, the source code of a program is leveraged to find every executable path. This helps to determine all the faults within a particular code.

EXPERIENCE BASED TESTING

These techniques are highly dependent on tester's experience to understand the most important areas of the software. The types of experience based testing are

Error Guessing

In this technique, the testers anticipate errors based on their experience, availability of data and their knowledge of product failure. Error guessing is dependent on the skills, intuition, and experience of the testers.

Exploratory Testing

This technique is used to test the application without any formal documentation. There is minimum time available for testing and maximum for test execution. In exploratory testing, the test design and test execution are performed concurrently.

MASTER TEST PLAN

The **master test plan** is a document that describes in detail how the testing is being planned and how it will be managed across different **test levels**.

It gives a bird's eye view of the key decisions taken, the strategies to be implemented and the testing effort involved in the project.

The master test plan for software testing provides the following details:

- List of tests to be performed
- Testing levels to be covered
- Relationship among different test levels and related coding activity
- Test Implementation Strategy
- Explain testing effort which is a component of the project
- Master test plan should align with test policy and test strategy. It should list any exceptions or deviations and their possible impact

MASTER TEST PLAN

The exact **structure of the master test plan** and its content depends on the following factors:

- Type of organization
- Documentation standards followed by the organization
- Level of project formality

TEST CASE MANAGEMENT

Test case management involves managing different versions of test cases, keeping track of changes in them, keeping a separate repository of test cases based on type of tests, as well as creating and managing automation scripts.

Understanding the Idea of a Test Case

Requirements tell you what the software needs to do.

Test scenarios define a means for confirming the requirement.

Test cases describe how, the component pieces of confirming that requirement.

Test scripts (whether automated or executed manually) tell you exactly how to execute that component piece.

Typically, you'll have a number of test cases per test scenario, covering various permutations of inputs and behaviors. You then have one or possibly more scripts per test case.

ANATOMY – TEST CASE

An ID: A unique way of identifying the test case.

Title or brief description: A quick means of understanding the software activity in question.

Related requirement and/or test: What broader scenario and requirement does this test case roll up to?

Remarks/Notes: Free form comments about the test case.

Script: Exact steps for executing the test.

Pass/Fail Status: Is the test case currently passing or failing?

History and Audit Trail: You should be able to see its pass/fail history as well as general changes and who has run/modified the test case.

TEST BED

Test bed preparation involves installing the application on a machine that is accessible to all test teams .

Care is taken to ensure that this machine is free of any interference from unauthorized access. Test data is populated in the application.

Care should also be taken to ensure that the test bed resembles the production environment as closely as possible, including all software and hardware configurations.

TEST CASE EXECUTION

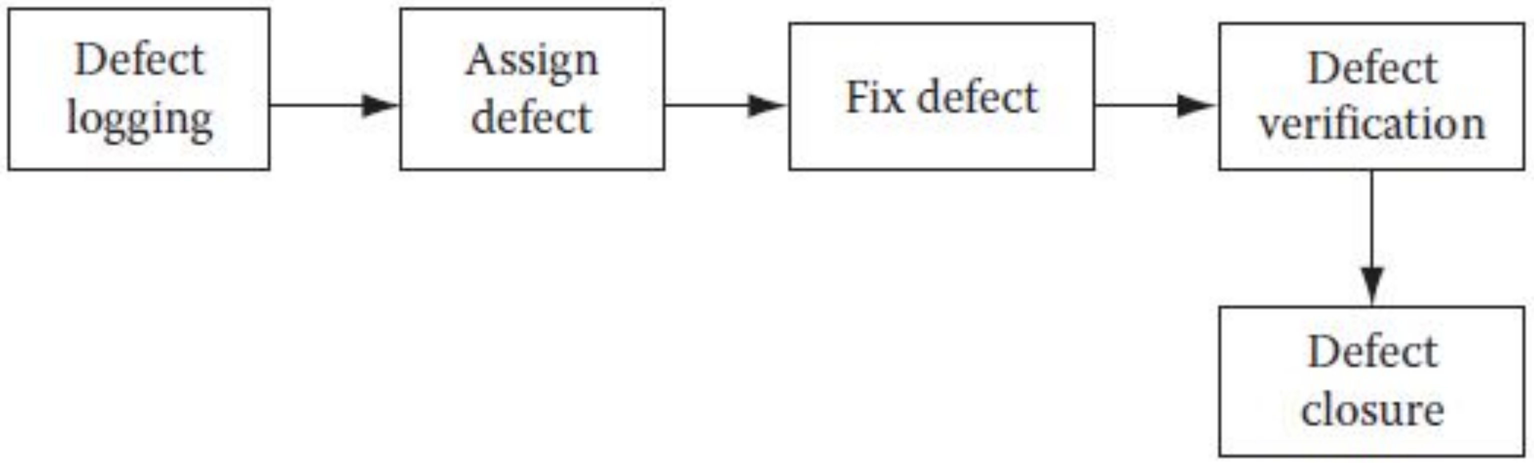
- Test case execution involves executing prepared test cases manually or using automation tools to execute them.
- For regression tests, automated test execution is a preferred method.
- After each test case is executed, it may pass or fail.
- If it fails then defects have to be logged.
- Exit criteria for test case execution cycle are generally defined in advance.
- Generally, when a certain level of quality of the application is reached, then test execution stops.

DEFECT TRACKING

Defect tracking is one of the most important activities in a test project

During defect tracking it is ensured that defects are logged and get fixed. All defects and their fixing are tracked carefully

Defect count per hour per day is a common way of measuring performance of a test team.



DEFECT TRACKING

If the testing is done for an in-house software product, traditionally, it used to not be a performance evaluation measurement.

A good defect tracking application should be deployed on a central server that is accessible to all test and development teams.

Each defect should be logged in such a way that it could be understood by both development and testing teams.

Generally, the defects should be reproducible, but in many instances, this is difficult

TEST REPORTING

During the execution of a test project, many initial and final reports are made.

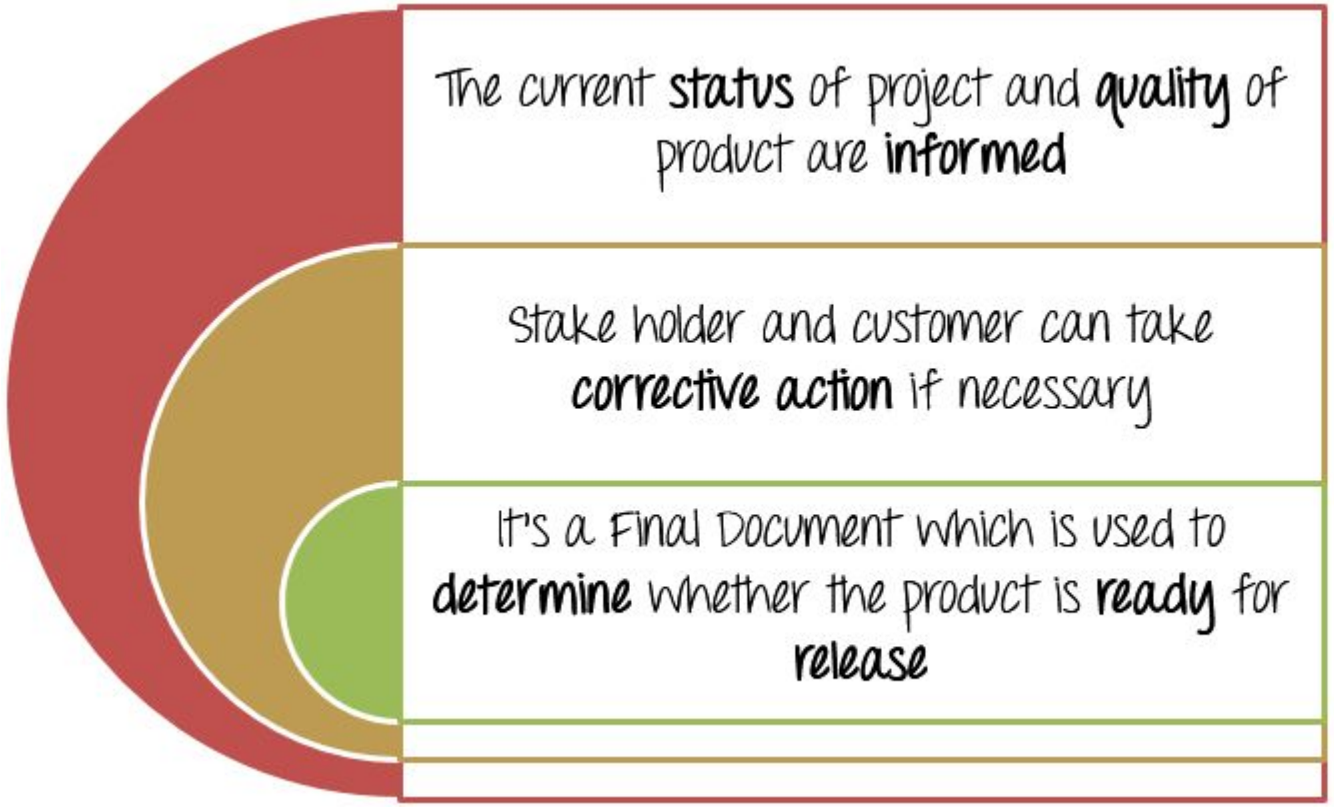
Test Report is a document which contains

A **summary** of test activities and final test results An **assessment** of how well the Testing is performed

Based on the test report, the stakeholders can **Evaluate** the **quality** of the tested product

Make a **decision** on the software release

TEST REPORTING

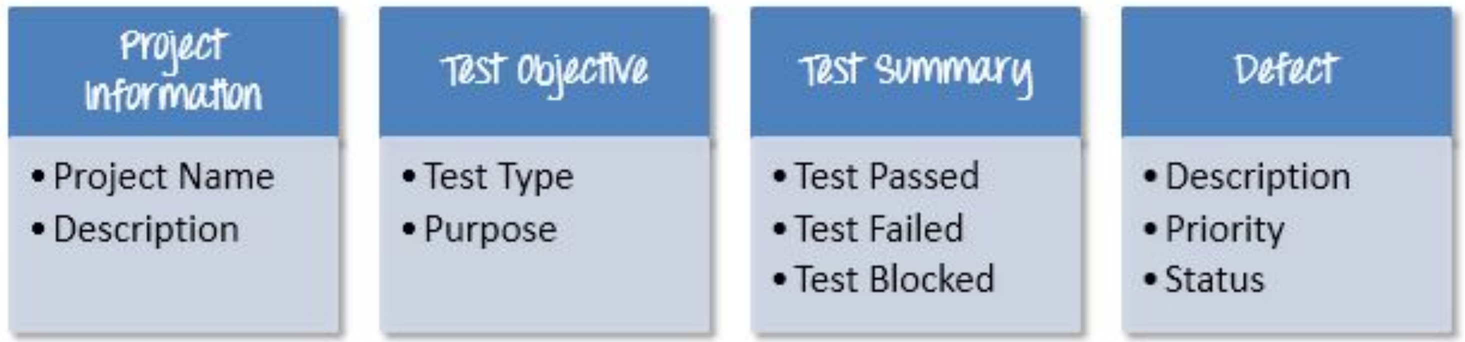


The current **status** of project and **quality** of product are **informed**

Stake holder and customer can take **corrective action** if necessary

It's a Final Document which is used to **determine** whether the product is **ready** for **release**

TEST REPORTING



Project Information
All information of the project such as the project name, product name, and version should be described in the test report.

Test Objective
Test Report should include the objective of each round of testing, such as Unit Test, Performance Test, System Test.. etc

TEST REPORTING

Test Summary

This section includes the summary of testing activity in general. Information detailed here includes

The number of test cases executed

The numbers of test cases pass

The numbers of test cases fail

Pass percentage

Fail percentage

Comments

This information should be displayed **visually** by using **color indicator**, **graph, and highlighted table**.

Defect

One of the most important information in Test Report is defect. The report should contain following information

Total number of bugs

Status of bugs (open, closed, responding)

Number of bugs open, resolved, closed

Breakdown by severity and priority

Like test summary, you can include some simple metrics like Defect density, % of fixed defects.

OTHER REPORTING DOCUMENTS

Test reports include test planning reports, test strategy reports, requirement document review comments, number of test cases created, automation scripts created, test execution cycle reports, defect tracking reports, etc.

Some other reports include traceability matrix reports, defect density, test execution rate, test creation rate, test automation script writing rate, etc.

TEST ARTIFACTS

An integral part of software **testing**, **test artifacts** are the various by-products generated during the process of software **testing**, which are then shared with the clients, team managers, team lead, and other team members and stakeholders associated with the project.

They include test plan document, test strategy document, test cases, test cycle logs, defect list, verification and validation reports, and product certification.

Management Artifacts

Customers are concerned not only with project cost and schedule, but they are also concerned with critical defects, which the test team has either detected or not.

So the management artifacts (metrics) include project cost compliance, project schedule compliance, and quality (number of critical defects caught versus number of critical defects which went into production).

MANAGEMENT RELATED - TEST ARTIFACTS

Management artifacts include traceability matrix, defect density rate, resource loading etc.

Requirement traceability matrix, as the name indicates is a matrix that contains tables which show many to many relationships between the requirement and the test cases.

The matrix shows which requirements are implemented by which one test case or test cases and which test case/ cases are mapped to which specific requirement or requirements

Defect Density - **Defect Density** is the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module.

It enables one to decide if a piece of software is ready to be released. **Defect density** is counted per thousand lines of code also known as KLOC