SCHOOL OF COMPUTER SCIENCE AND IT

School of Computer Science & IT

Department of BCA

SOFTWARE ENGINEERING (22BCA3C01)

MODULE 5: Software Testing

Software Testing: Basics

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data (called test data).
- You check the results of the test for errors, anomalies or information about the program’s non-functional attributes.
- Can reveal the presence of errors, NOT the absence of errors. That is why we do enough testing, but exhaustive testing may not be possible.
- Testing is part of a more general verification and validation process.

Verification vs validation

- **Testing Goals**:

1. To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification. This is Verification.
2. To demonstrate to the developer and the system customer that the software meets its requirements. This is called Validation.

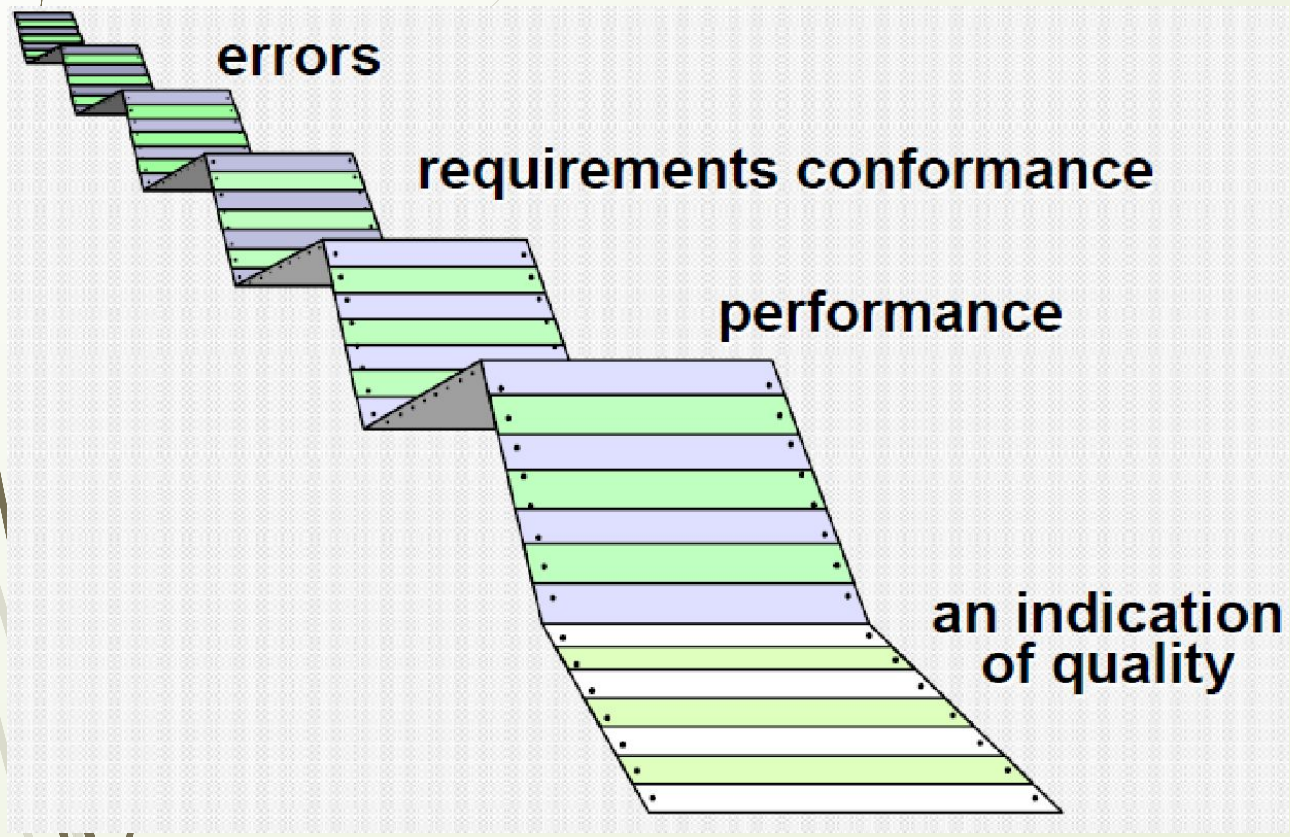
- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function. The test cases are designed to expose defects.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. You expect the system to perform correctly using a given set of test cases that reflect the system’s expected use.

- Barry Boehm, an pioneer in software engineering states this another way:
- **Verification**: "Are we building the product right”. i.e., The software should conform to its specification.
- **Validation**: "Are we building the right product”. i.e., The software should do what (specific functions) the user really requires.

Difference between Verification and Validation

Verification	Validation
Verification is the process to find whether the software meets the specified requirements for particular phase.	The validation process is checked whether the software meets requirements and expectation of the customer.
It estimates an intermediate product.	It estimates the final product.
The objectives of verification is to check whether software is constructed according to requirement and design specification.	The objectives of the validation is to check whether the specifications are correct and satisfy the business need.
It describes whether the outputs are as per the inputs or not.	It explains whether they are accepted by the user or not.
Verification is done before the validation.	It is done after the verification.
Plans, requirement, specification, code are evaluated during the verifications.	Actual product or software is tested under validation.
It manually checks the files and document.	It is a computer software or developed program based checking of files and document.

What Testing Shows



- Tests are executed to uncover errors in meeting customer requirements.
- As errors are uncovered, they must be debugged to achieve requirements conformance.
- Requirements conformance leads to overall software performance.
- Improvement of software performance is an indication of quality software.

Strategic Approach

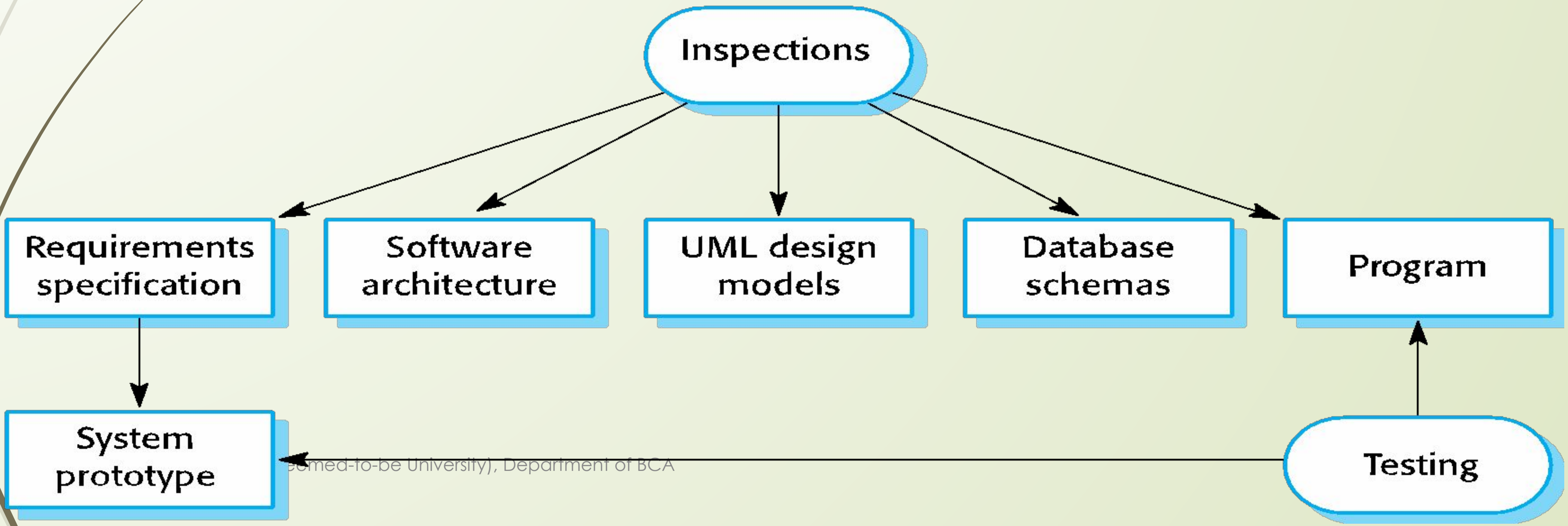
- To perform effective testing, you should conduct effective technical reviews (also called Inspections). By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities; while testing finds errors, debugging removes errors. Debugging must be accommodated in any testing strategy.

Inspections and testing

Software inspections concerned with analysis of the static system representation to discover problems (static verification)

Software testing concerned with exercising and observing product behaviour (dynamic verification)

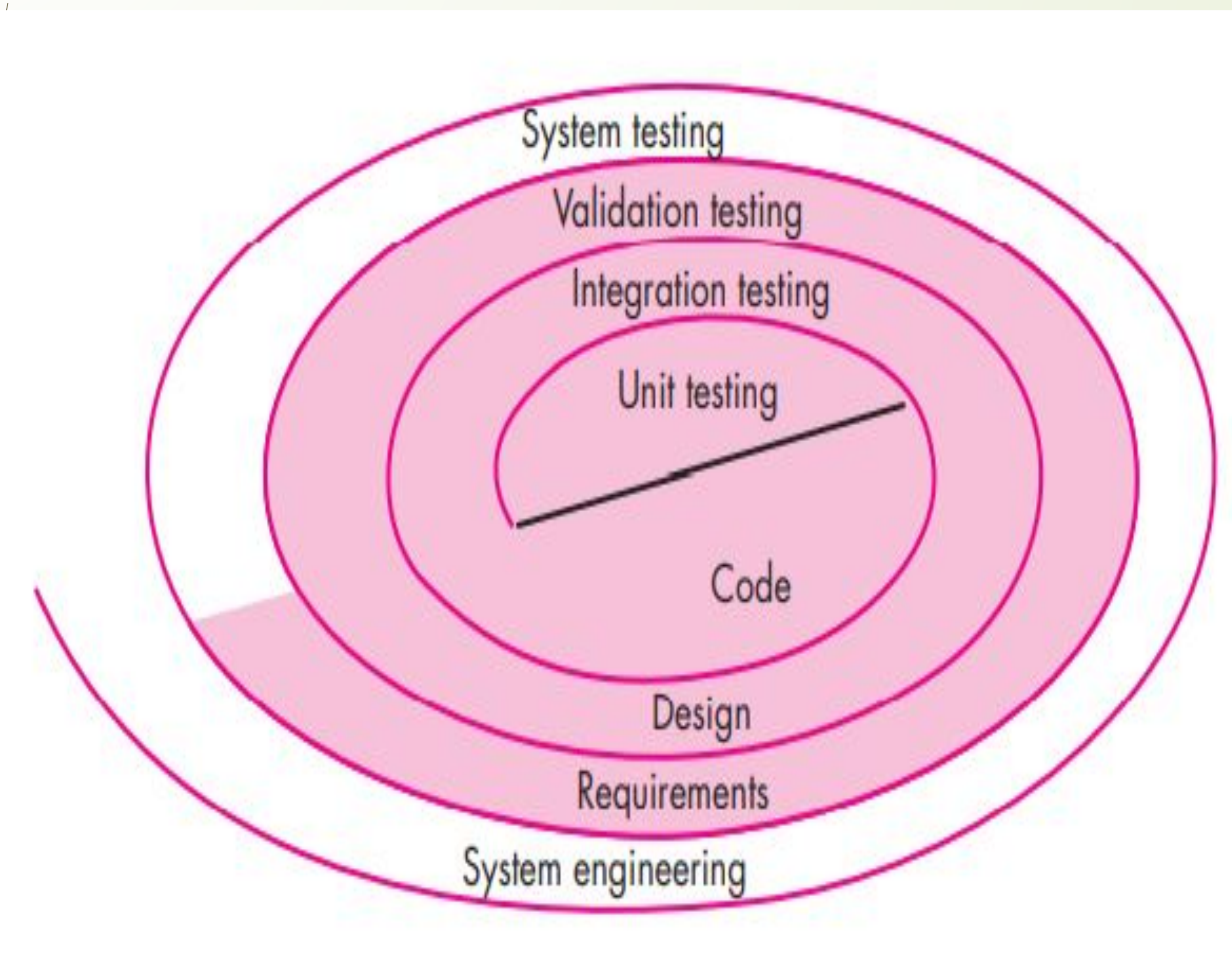
The system is executed with test data and its operational behaviour is observed.



Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, designs, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.
- Inspections and testing are complementary and not opposing verification techniques.
- 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.

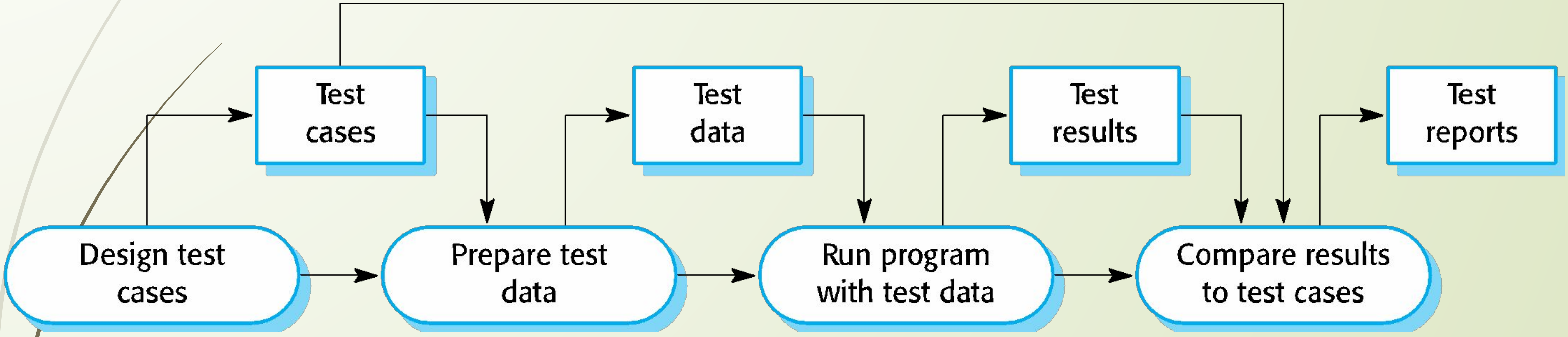
Testing Strategy



- 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
  - focus on class that encompasses attributes and operations and implies communication and collaboration
  - Integrate classes one after another to form the whole system.

A model of the software testing process

- **Test Case: A test case is a specification of the inputs, execution conditions, testing procedure, and expected results that defines a single test to be executed to achieve a particular software testing objective.**



Sample Test Case

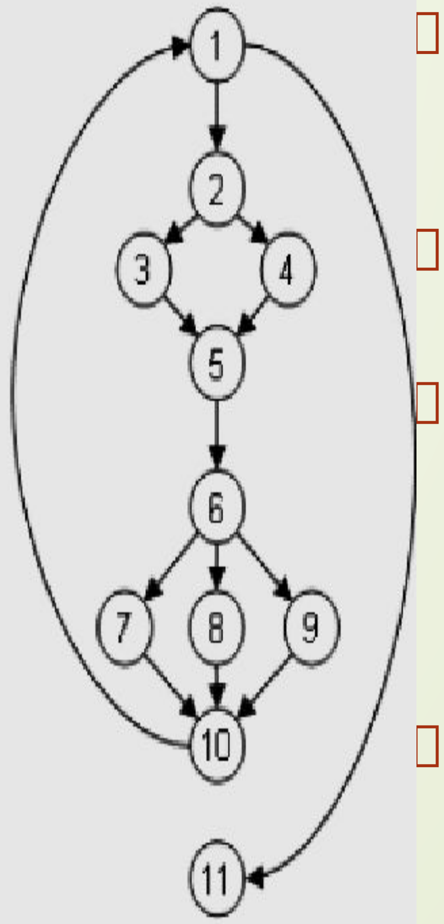
Test case ID	Test case description	Prerequisites	Test steps	Test data	Expected Result	Actual Result	Status	Created By	Date of creation	Executed By	Date of execution
TC001	The objective of this test case is to verify the 'Login'	1. User is authorized 2. Has an account	1. Enter valid username 2. Enter valid password 3. Click on 'Login' button	1.valid user name and pass word 2. invalid user name and pass word	1. User should be able to login his Gmail account with his valid credentials 2. 'Invalid username or password' should get displayed if the username and password are not valid	1. If the valid credentials are entered then the user will be able to login his / her account 2. If invalid credentials are entered then nothing happens(the expected message is not displayed)	Fail	Rajesh	1/1/2016	Umesh	1/2/2016

Session -23

- Software Testing Strategies
  - For Conventional Software
    - Unit Testing
    - Integration Testing
  - Regression Testing
  - Smoke Testing

Unit Testing

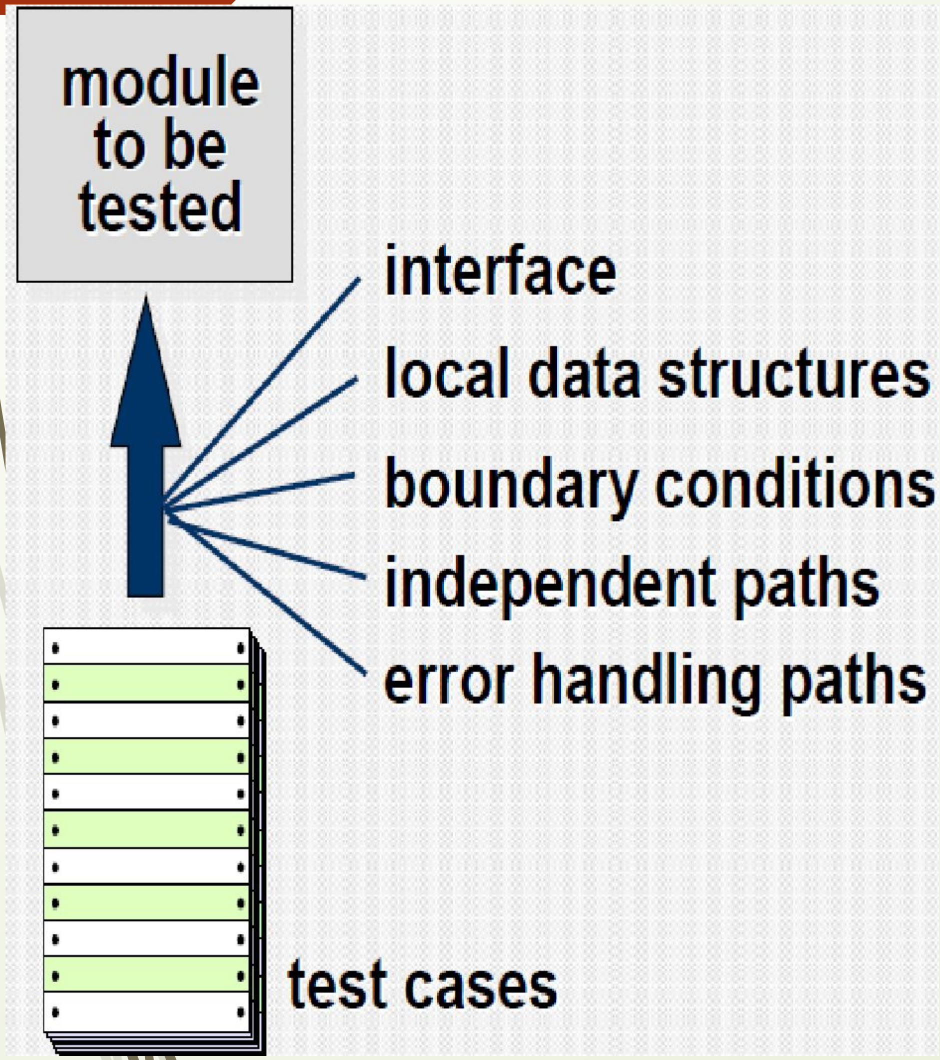
Node	Statement
(1)	`while(x<100){`
(2)	`if (a[x] % 2 == 0) {`
(3)	`parity = 0;`
	`}`
	`else {`
(4)	`parity = 1;`
(5)	`}`
(6)	`switch(parity){`
	`case 0:`
(7)	`println( "a[" + i + "] is even");`
	`case 1:`
(8)	`println( "a[" + i + "] is odd");`
	`default:`
(9)	`println( "Unexpected error");`
	`}`
(10)	`x++;`
	`}`
(11)	`p = true;`



The Cyclomatic complexity of the program can be defined using the formula – V(G) = E - N + 2 = 14-11 +2= 5

- Unit testing is the process of testing individual components in isolation.
- It focuses testing on the function or software module of conventional software.
- It concentrates on the internal processing logic and data structures.
- Rule of Thumb: Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources (manpower, time etc.) are limited.
- Cyclomatic complexity is **a software metric used to indicate the complexity of a program**. It is a quantitative measure of the number of linearly independent paths through a program's source code.

Targets for Unit Test Cases

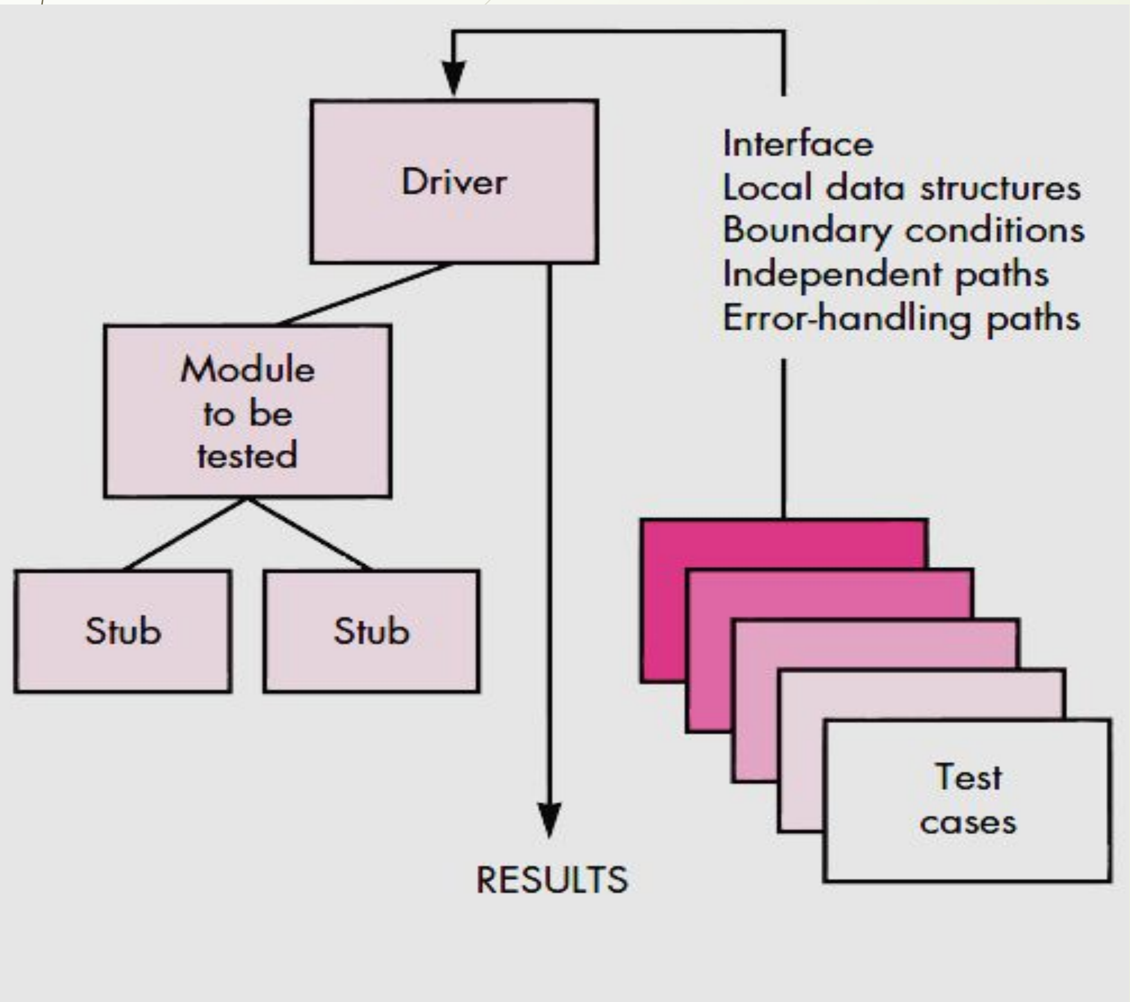


- Module interface
  - Ensure that information flows properly into and out of the module.
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution.
- Boundary conditions
  - Ensure that the module operates properly at boundary 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.

Common Computational Errors in Execution Paths

- 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).
- Comparison of different data types.
- Incorrect logical operators or precedence.
- Incorrect comparison of variables.
- Improper or nonexistent loop termination.
- Improperly modified loop variables.
- Boundary value violations.

Unit Test Procedure: Drivers and Stubs for Unit Testing



- Because a component/ module is not stand-alone program, **driver and or stub** software must be developed for each unit under test.
- **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.

Stubs and Drivers

- The Stubs and Drivers are considered as elements that are equivalent to to-do modules that could be replaced if modules are in their developing stage, missing or not developed yet so that the necessity of such modules could be met. Drivers and stubs simulate features and functionalities and have the ability to serve features that a module can provide. This reduces useless delays in testing and makes the testing process faster.
- Stubs are mainly used in Top-Down integration testing while the Drivers are used in Bottom-up integration testing, thus increasing the efficiency of the testing process.

What are Stubs?

- Stubs are developed by software developers to use them in place of modules, if the respective modules aren't developed, missing in developing stage, or are unavailable currently while Top-down testing of modules. A Stub simulates module which has all the capabilities of the unavailable module. Stubs are used when the lower-level modules are needed but are unavailable currently.

Stubs are divided into four basic categories based on what they do :

- Shows the traced messages,
- Shows the displayed message if any,
- Returns the corresponding values that are utilized by modules,
- Returns the value of the chosen parameters(arguments) that were used by the testing modules.

What are Drivers?

Drivers serve the same purpose as stubs, but drivers are used in Bottom-up integration testing and are also more complex than stubs. Drivers are also used when some modules are missing and unavailable at time of testing of a specific module because of some unavoidable reasons, to act in absence of required module. Drivers are used when high-level modules are missing and can also be used when lower-level modules are missing.

Ex : Suppose, you are told to test a website whose corresponding primary modules are, where each of them is interdependent on each other, as follows:

Module-A : Login page website,

Module-B : Home page of the website

Module-C : Profile setting

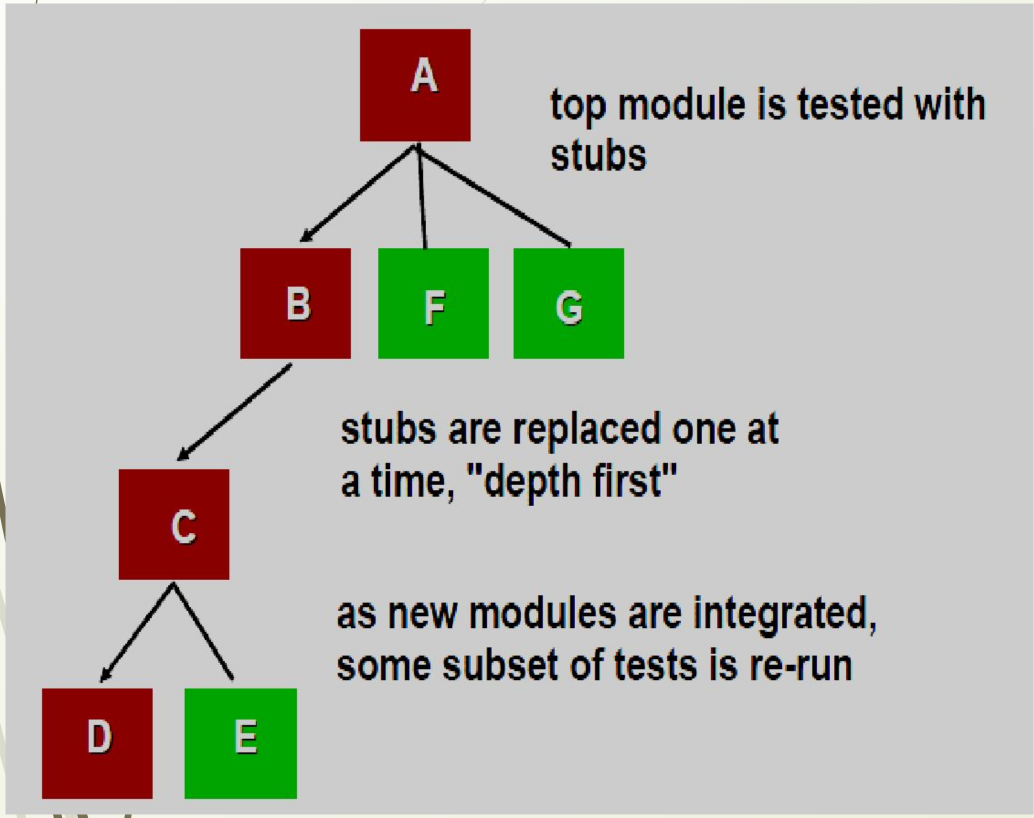
Module-D : Sign-out page

- It's always considered good practice to begin development of all modules parallelly because as soon as each gets developed they can be integrated and could be tested further as per their corresponding interdependencies order with a module. But in some cases, if any one of them is in developing stage or not available in the testing process of a specific module, stubs or drivers could be used instead.
- Assume **Module-A** is developed. As soon as it's developed, it undergoes testing, but it requires **Module-B,** which isn't developed yet. So in this case, we can use the **Stubs or Drivers** that simulate all features and functionality that might be shown by actual **Module-B.** So, we can conclude that Stubs and drivers are used to fulfill the necessity of unavailable modules. Similarly, we may also use Stubs or Drivers in place of **Module-C** and **Module-D** if they are too not available.

Integration Testing

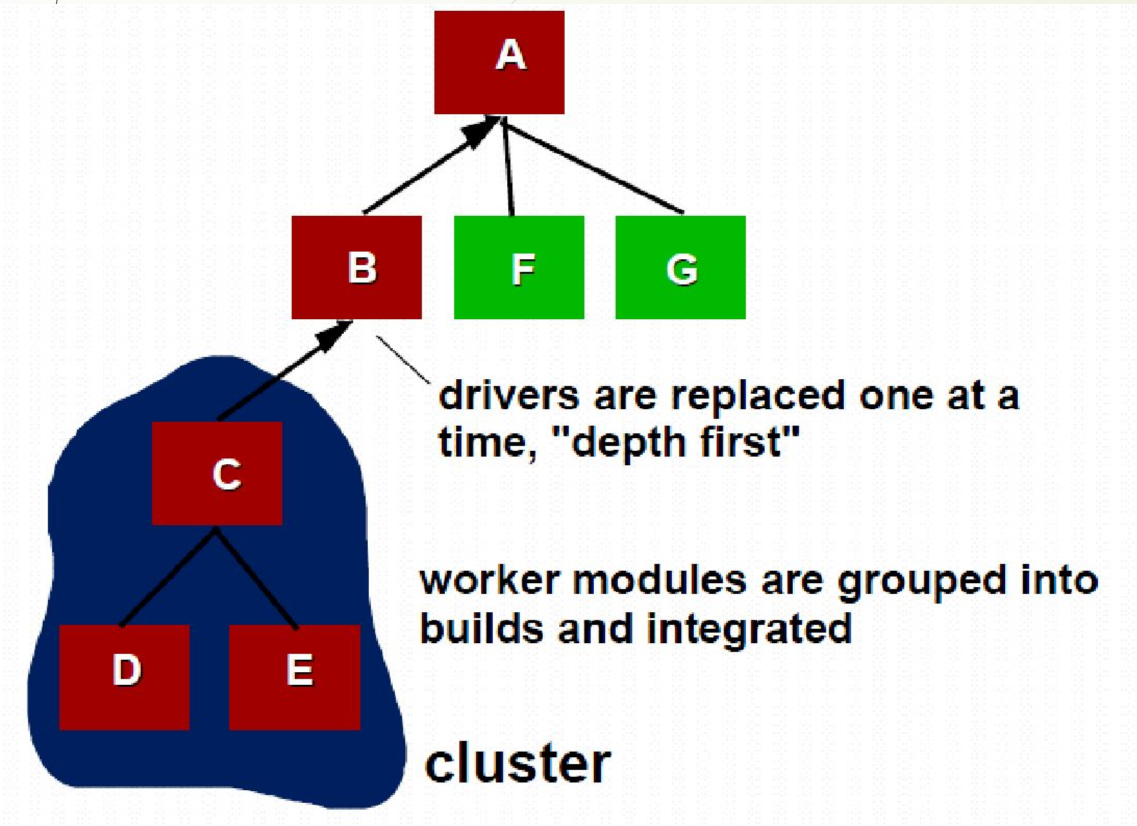
- Unit tested modules are combined one at a time to build a larger structure of the software system based on the prescribed design. Module integration is done incrementally.
- While modules are integrated, tests are conducted to uncover errors associated with the integration.
- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration

Top-down Integration



- Modules are integrated by moving downward through the control hierarchy, beginning with the main module.
- 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.
- **Advantages**
  - This approach verifies major control or decision points early in the test process.
- **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. This may require more testing when lower level modules are integrated.

Bottom-up Integration



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

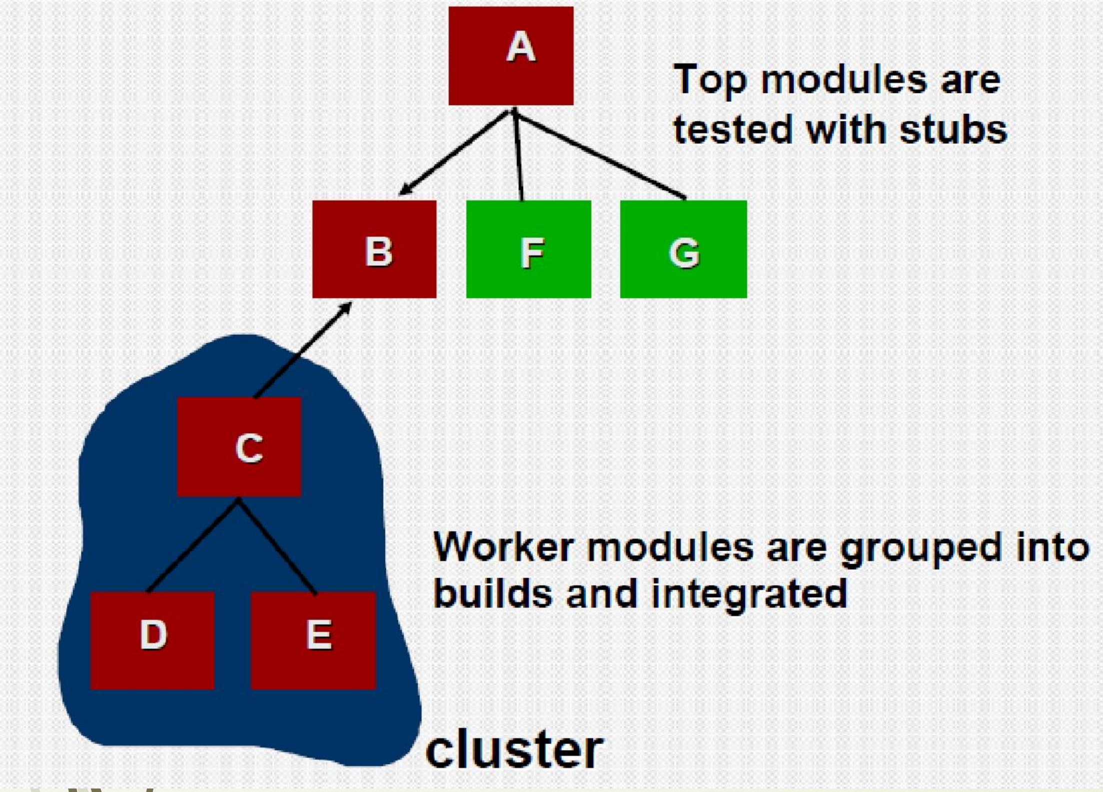
Advantages

- 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.
- **Advantage:** Reaps the advantages of both types of integration while minimizing the need for drivers and stubs.
- **Disadvantage:** Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario.

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 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.

Smoke Testing

Allows the software team to assess its project on a frequent basis. 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.

Benefits of Smoke Testing

- 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.

Session -24

- Software Testing Techniques
  - White-box testing vs Black-box testing
    - Basis Path Testing

Two categories of Testing Techniques

- 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/or loops.
- 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 .

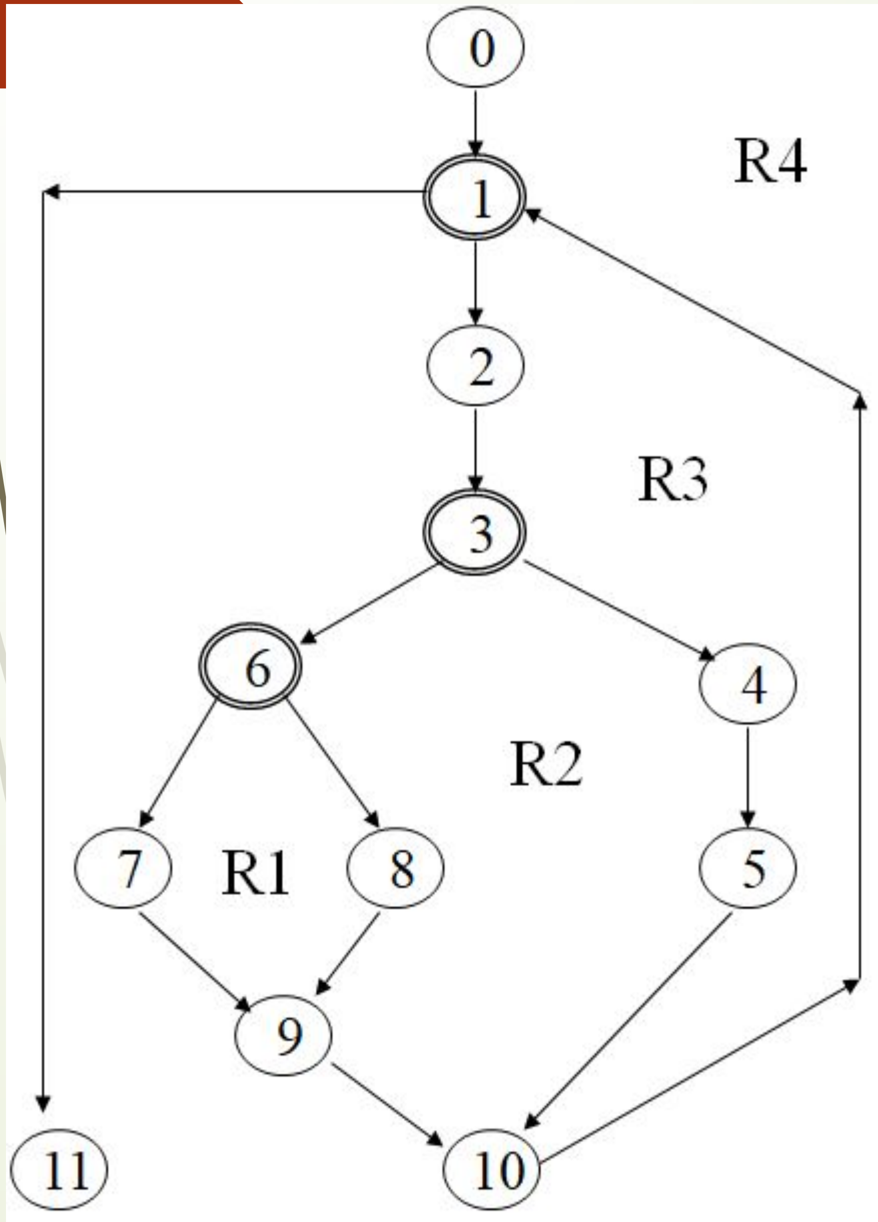
White-box Testing

- White-box testing sometime referred to as **Glass-box testing** is an approach used to test the internal logic / procedure of the module.
- Using the approach, a software engineer achieves the following:
  - 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 (simple and nested) at their boundaries and within their operational bounds.
  - Exercise internal data structures to ensure their validity.

Basis Path Testing

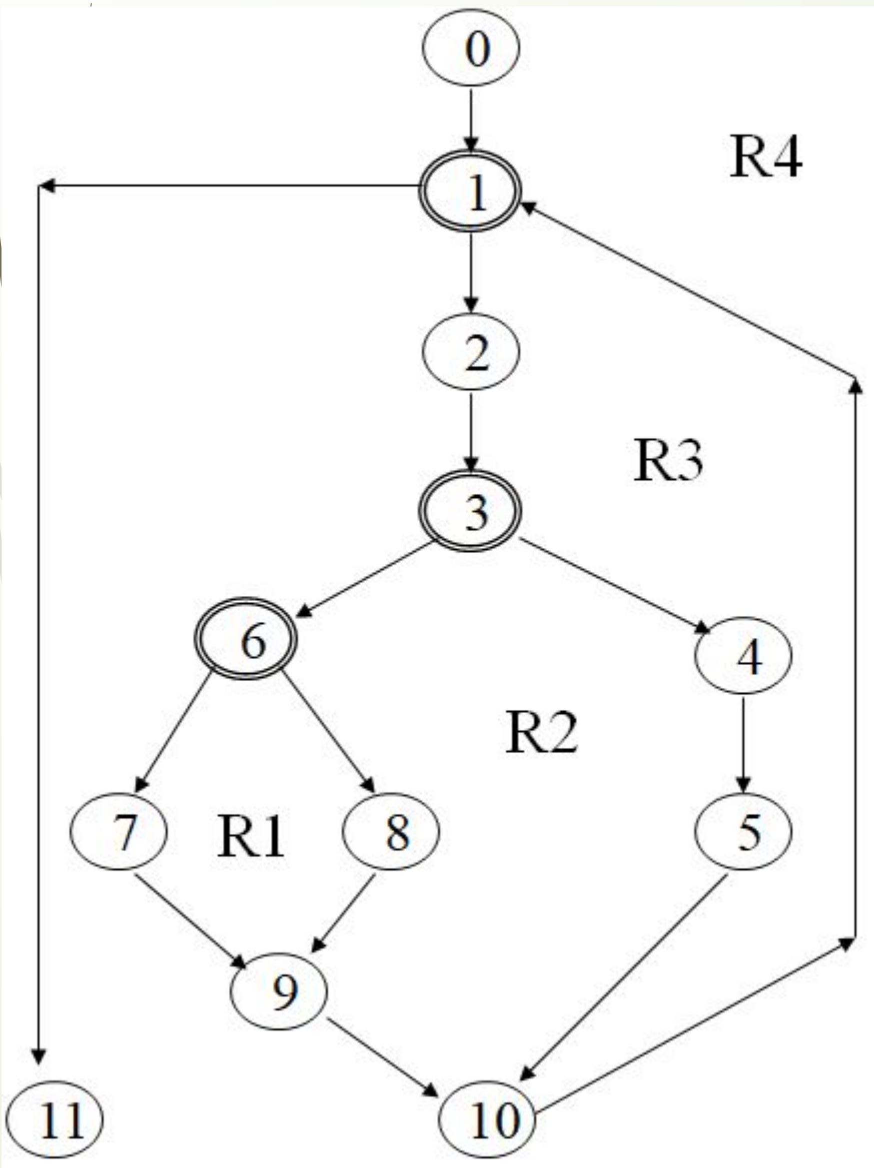
- This is a White-box testing technique proposed by Tom McCabe.
- It enables the test case designer to derive a logical complexity measure of a procedural design.
- It uses this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.
- A Flow Graph is constructed to show the basis paths.

Flow Graph Notation



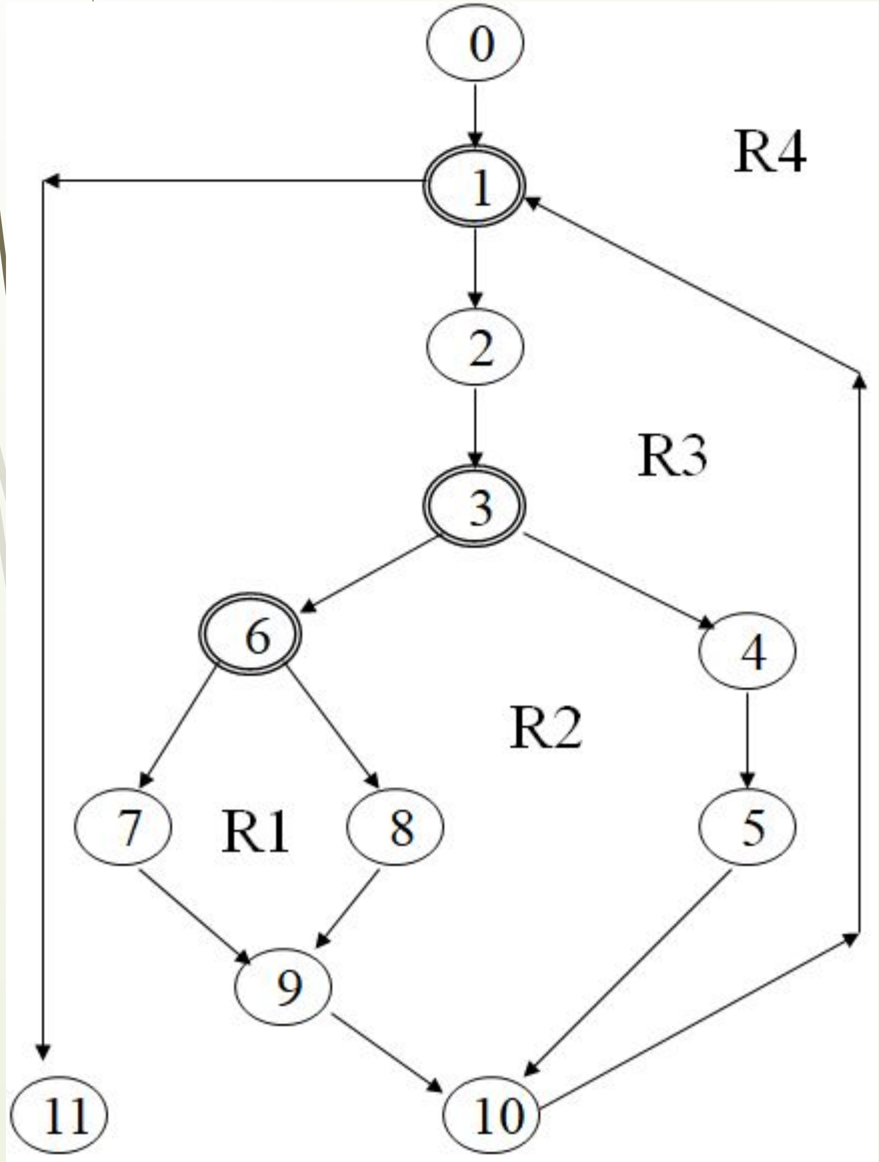
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements.
- A node containing a simple conditional expression is referred to as a predicate node.
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, 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.
- 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.

Basis Paths / Independent 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
  - 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

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,**

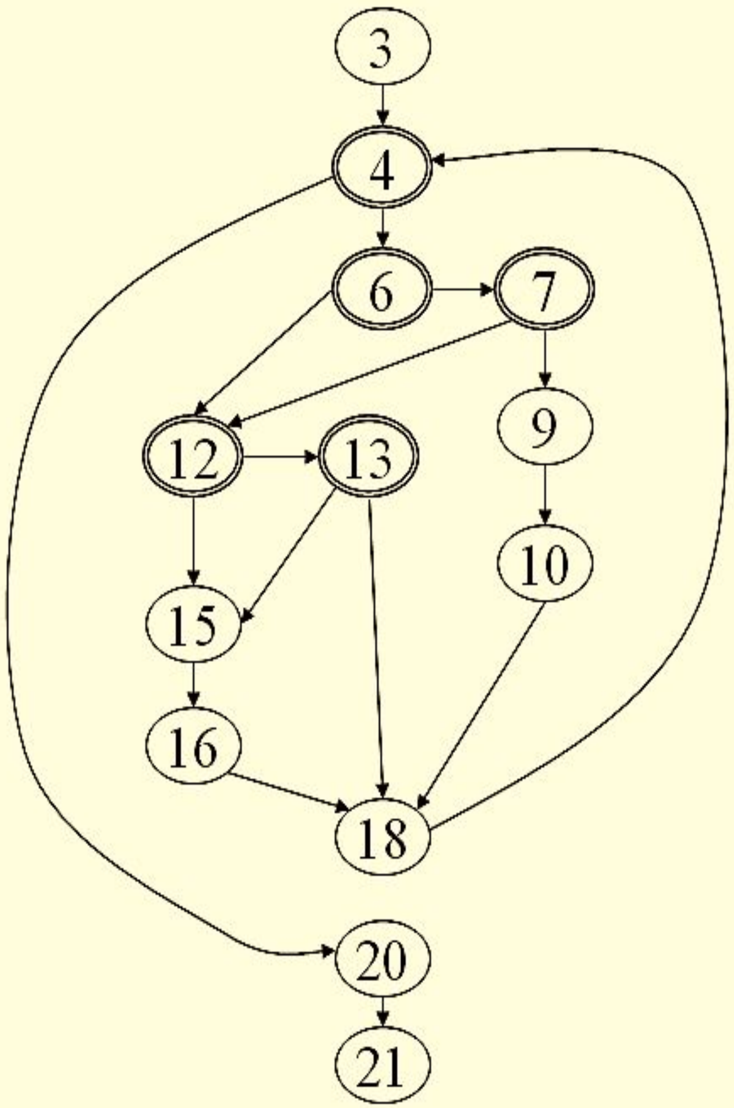
  - where E is the number of edges and N is the number of nodes in graph G.
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G.
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Deriving the Basis Set and Test Cases

```
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
```



1) Using the code, 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.

V(G) = 17 edges – 13 nodes + 2 = 6

Basis Set:

- Path 1: 3-4-20-21
- Path 2: 3-4-6-12-15-16-18-4-20-21
- Path 3: 3-4-6-12-13-15-16-18-4-20-21
- Path 4: 3-4-6-12-13-18-4-20-21
- Path 5: 3-4-6-7-12-13-18-4-20-21
- Path 6: 3-4-6-7-9-10-18-4-20-21

School of Computer Science & IT

Department of BCA

SOFTWARE ENGINEERING (22BCA3C01)

MODULE 4: Software Testing

Session -25

- Software Testing Techniques
    - Black-box Testing
        - Equivalence Partitioning
        - Boundary Value Analysis
    - Object-Oriented Testing Techniques

Black-box Testing

- Here, the tester has no knowledge of the internal logic of the program/ software.
- It complements white-box testing by uncovering different types of errors.
- Used after white box testing has been performed.
- Focuses on the functional requirements and the information domain of the software.
- The test cases uncover the following errors:
  - Incorrect or missing functions.
  - Interface errors.
  - Errors in data structures or external data base access.
  - Behavior or performance errors.
  - Initialization and termination errors.

Equivalence Partitioning

- **Equivalence Partitioning** or Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc.
- In this technique, the input domain of a program is divided into equivalence data partitions (**also called equivalence class**) that can be used to derive test cases which reduces time required for testing because of small number of test cases are needed.
- 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.

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 values x}
- If an input condition is a Boolean value, one valid and one invalid class are defined
  - Input: {true condition} Eq classes: {true condition}, {false condition}

Boundary Value Analysis

- It is another frequently used black-box testing technique.
- 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 an input domain.

Guidelines for Boundary Value Analysis

- **1.** If an input condition specifies a range 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*
  - Consider an example where a program under test accepts values only from 100 to 5000.
  - The valid and invalid test cases are listed below.
  - **Valid Test Cases**

    a) Enter the value 100 which is min value.

    b) Enter the value 101 which is min+1 value.

    c) Enter the value 4999 which is max-1 value.

    d) Enter the value 5000 which is max value.
  - **Invalid Test Cases**

    a) Enter the value 99 which is min-1 value.

    b) Enter the value 5001 which is max+1 value
- **2.** If an input condition specifies a number of values (i.e. a set), 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

Testing for Object-Oriented Software

- Conventional test case design is driven by an input-process-output view of software.
- Object-oriented testing focuses on designing appropriate sequences of methods to exercise the states of a class.
- Because attributes and methods are encapsulated in a class, testing methods from outside of a class is generally unproductive.
- Inheritance requires retesting of each new context of usage for a class.
  - If a subclass is used in an entirely different context than the super class, the super class test cases will have little applicability and a new set of tests must be designed.
- Conventional Testing can also be applied to Object Oriented Systems
- White-box testing can be applied to the operations /methods defined in a class.
  - Basis path testing and loop testing can help ensure that every statement in an method has been tested.
- Black-box testing methods are also appropriate.
  - Use cases can provide useful input in the design of black-box tests.

Class Level Testing: Random Testing

- Certain methods in a class may constitute a minimum behavioral life history of an object; consequently, they may have implicit order dependencies or sequences designed into them.
- For example,
  - In File or Database operations: open, read, write, close;
  - In Bank Account operations: open, deposit, withdrawal, balance, close.
- Using the methods for a class, a variety of method sequences are generated randomly and then executed.
- The goal is to detect these order dependencies or sequences and make appropriate adjustments to the design of the methods.

Class Level Testing: Partition Testing

- *Partition testing* reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for conventional software.
- Methods are grouped based on one of three partitioning approaches: state-based, attribute-based, category-based.
- State-based partitioning categorizes class methods based on their ability to change the state of the class.
  - Tests are designed in a way that exercise methods that change state and those that do not change state.
- Example:
  - Consider the Bank **Account** class, state operations include *deposit* and *withdraw*, whereas nonstate operations include *checkBalance, checkCreditLimit and generateStatement.*

- Attribute-based partitioning categorizes class methods based on the attributes that they use.
  - Methods are partitioned into those that read an attribute, modify an attribute, or do not reference the attribute at all.
- Example:
  - For the Bank **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 categorizes class methods based on the generic function that each performs.
  - Method categories can be initialization methods, computational methods, and termination methods.
- Example
  - Operations in the Bank **Account** class can be categorized in initialization operations (*open*), computational operations (*deposit, withdraw*). queries (checkB*alance, generateStatement, checkCreditLimit*) and termination operations (*close*).
  - Operations for Database access: connection opening method, data manipulation methods, connection close method.

Session -26

- Higher Order Testing
- Debugging Process

Higher Order Testing

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

Validation Testing

- Validation testing follows integration testing
- 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
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., 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

Types of user testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer’s site.
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing

  Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom software systems.

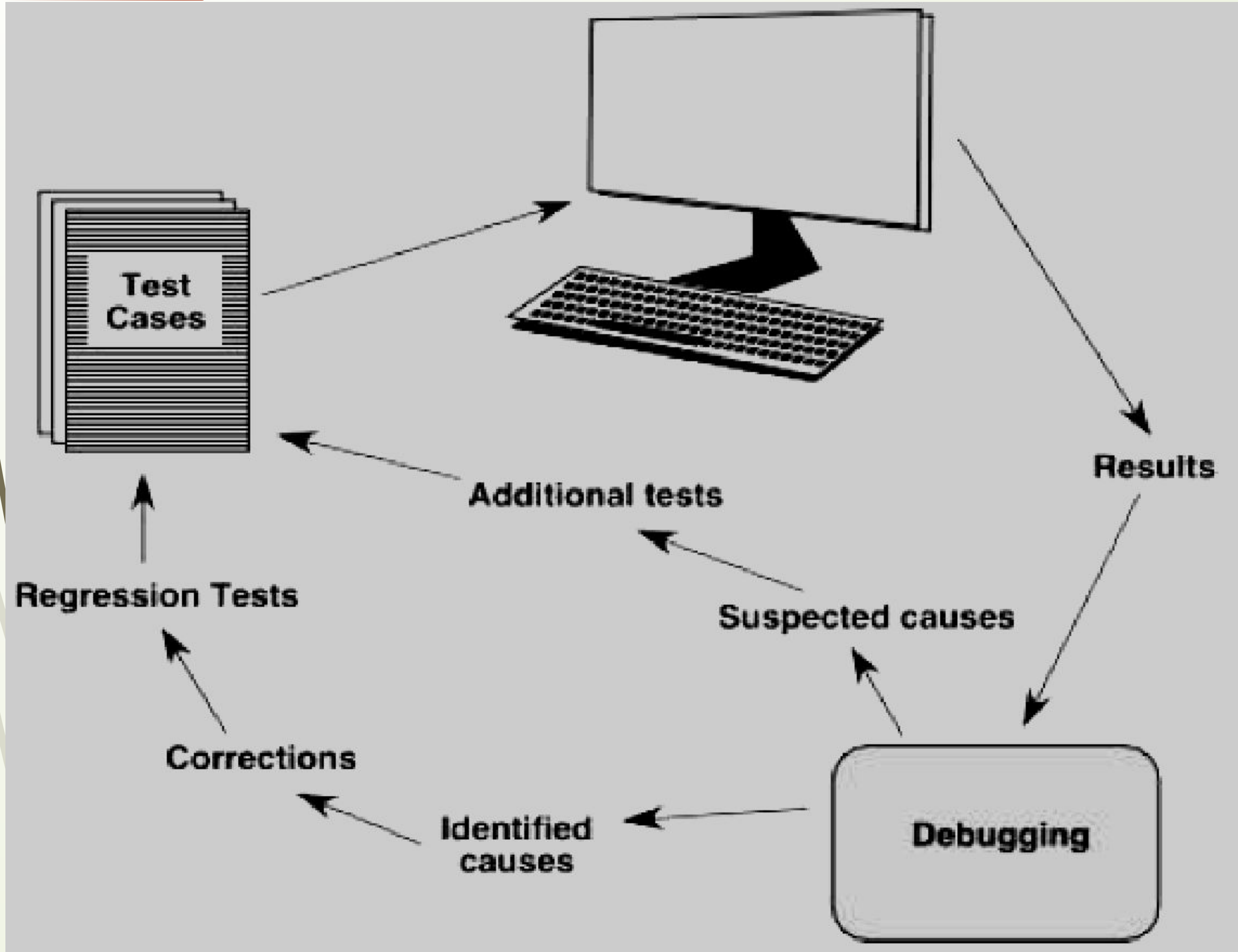
Alpha and Beta Testing

- Alpha testing
  - Conducted at the developer's site (may be 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

System Testing

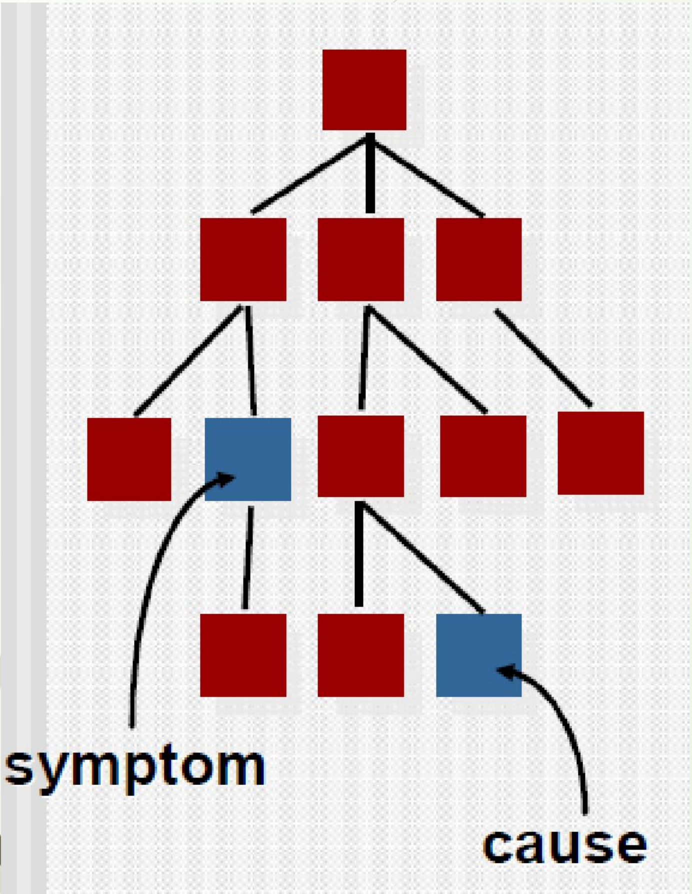
- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

Debugging Process



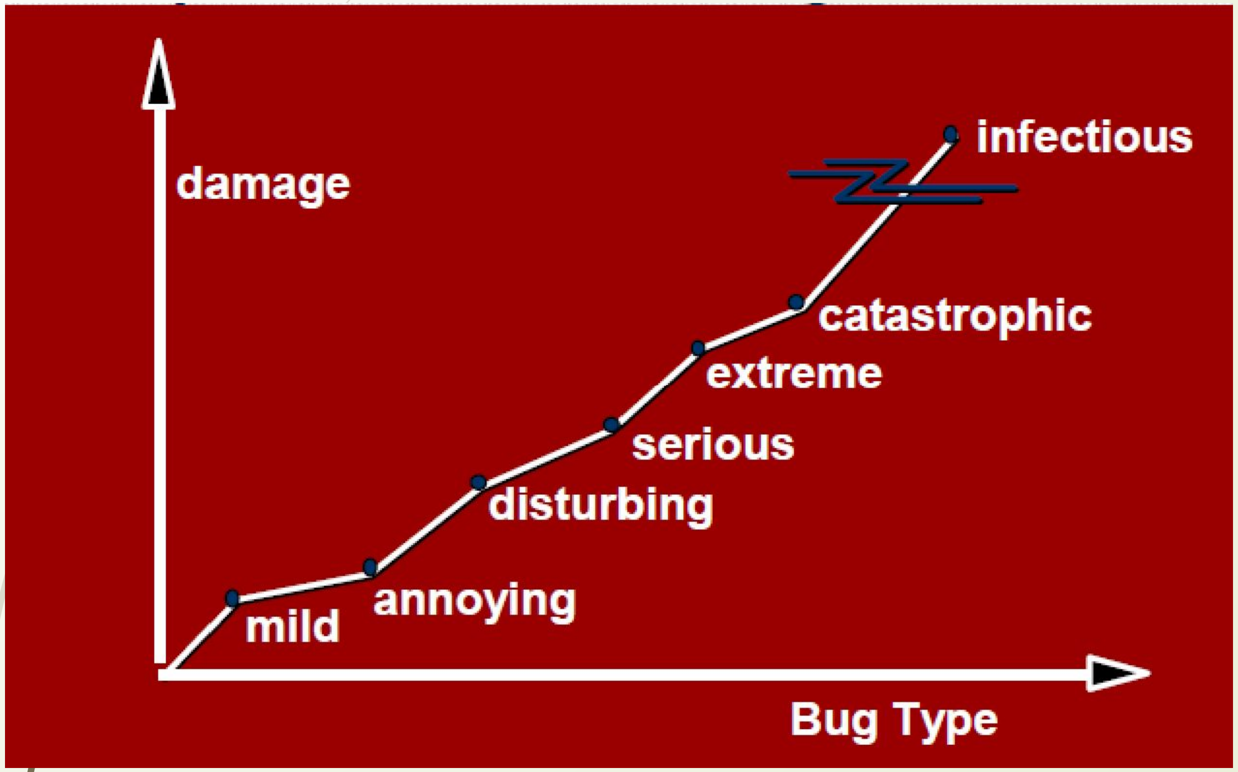
- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Why Debugging is so Difficult - Cause & Symptom



- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors (e.g., round-off accuracies)
- The symptom may be caused due to a system or compiler error
- The symptom may be caused by human error that is not easily traced
- The symptom may be caused due to assumptions that everyone believes
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

Consequences of Bugs



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

Debugging Techniques

- Objective of debugging is to find and correct the cause of a software error
- Bugs can be found by a combination of systematic evaluation, and testing
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging techniques
  - Brute force
  - Backtracking
  - Cause elimination

Strategy #1: Brute Force

- This is conceptually the simplest of the methods, and often the least successful.
- This involves the developer manually searching through stack-traces, memory-dumps, log files, and so on, for traces of the error.
- Extra output statements, in addition to break points, are often added to the code in order to examine what the software is doing at every step.

Strategy #2: Backtracking

- An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went wrong.
- In other words, start at the point where the program gives the incorrect result—such as where incorrect data were printed, program produces observable error. The developer than backtracks through the execution path, looking for the cause.
- Can be used successfully in small programs
- In large programs, the number of potential backward paths may become unmanageably large

Strategy #3: Cause Elimination

- In this method, the developer develops hypotheses as to why the bug has occurred. The code can either be directly examined for the bug, or data to test the hypothesis can be constructed.
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
- A list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug