- Once standard solutions emerge, no modifications to the program parts may be necessary. One can directly use the parts to develop his application. Reuse without modification is more useful than the classical library modules.

10. SOFTWARE TESTING

Software testing is a process of executing a program or application with the intent of finding the software bugs.

- ✓ It can also be stated as the **process of validating and verifying** that a software program or application or product:
  - Meets the business and technical requirements that guided it's design and development
  - Works as expected
  - Can be implemented with the same characteristic.

- ✓ **Verification:** The set of activities that ensure that software correctly implements a specific function or algorithm. (Are the algorithms coded correctly?) (Are we building the product right?)
- ✓ **Validation:** The set of activities that ensure that the software that has been built is traceable to customer requirements. (Does it meet user requirements?) (Are we building the right product?)

NEED OF SOFTWARE TESTING

1. Software testing is really required to point out the defects and errors that were made during the development phases.
2. It's essential since it makes sure of the Customer's reliability and their satisfaction in the application.
3. It is very important to ensure the Quality of the product. Quality product delivered to the customers helps in gaining their confidence.
4. Testing is necessary in order to provide the facilities to the customers like the delivery of high quality product or software application which requires lower maintenance cost and hence results into more accurate, consistent and reliable results.
5. Testing is required for an effective performance of software application or product.
6. It's important to ensure that the application should not result into any failures because it can be very expensive in the future or in the later stages of the development.
7. It's required to stay in the business.

TERMINOLOGIES IN TESTING

- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.
- **Error:** Error is deviation from actual and expected value. It represents mistake made by people.
- **Fault:** Fault is incorrect step, process or data definition in a computer program which causes the program to behave in an unintended or unanticipated manner. It is the result of the error.

- **Bug:** Bug is a fault in the program which causes the program to behave in an unintended or unanticipated manner. It is an evidence of fault in the program.

- **Failure:** Failure is the inability of a system or a component to perform its required functions within specified performance requirements. Failure occurs when fault executes.

- **Defect:** A defect is an error in coding or logic that causes a program to malfunction or to produce incorrect/unexpected results. A defect is said to be detected when a failure is observed.

OBJECTIVES OF SOFTWARE TESTING

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

COMMON ERRORS TO UNCOVER DURING 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)
- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

TESTING ACTIVITIES

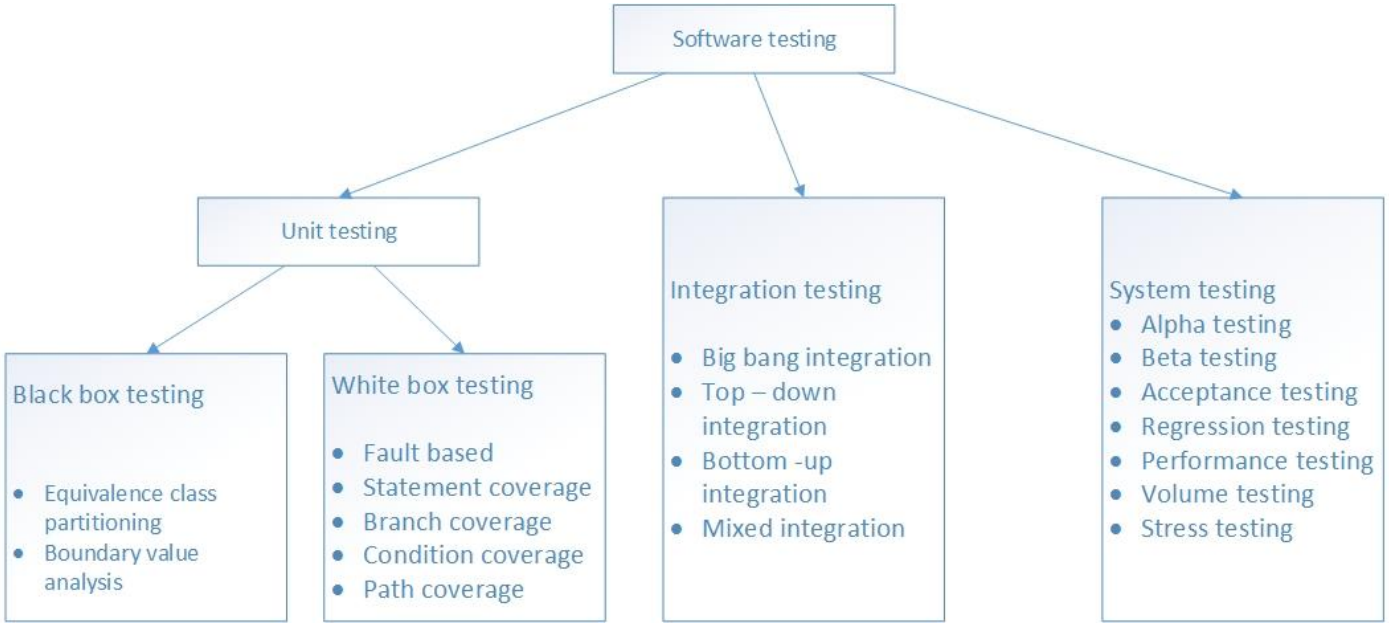
1. **Test Suite design**

2. **Running test cases and checking the result to detect failures*:*** *Each test case is run and the results are compared with expected results.*

3. **Debugging:** *To identify the statements that are in error.*

4. **Error correction:** *Code is appropriately changed to correct the error.*

TYPES OF TESTING



UNIT TESTING

- A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.
- **Unit tests are basically written and executed by software developers** to make sure that code meets its design and requirements and behaves as expected.
- The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly.
- This means that for any function or procedure when a set of inputs are given then it should return the proper values. It should handle the failures gracefully during the course of execution when any invalid input is given.
- Unit testing should be done before Integration testing. Unit testing should be done by the developers.
- A unit test provides a written contract that the piece of code must assure. Hence it has several benefits.

Advantages of Unit testing:

1. Issues are found at early stage. Since unit testing are carried out by developers where they test their individual code before the integration. Hence the issues can be found very early and can be resolved then and there without impacting the other piece of codes.

2. Unit testing helps in maintaining and changing the code. This is possible by making the codes less interdependent so that unit testing can be executed. Hence chances of impact of changes to any other code gets reduced.

3. Since the bugs are found early in unit testing hence it also helps in reducing the cost of bug fixes. Just imagine the cost of bug found during the later stages of development like during system testing or during acceptance testing.

4. Unit testing helps in simplifying the debugging process. If suppose a test fails then only latest changes made in code needs to be debugged.

BLACK BOX TESTING / FUNCTIONAL TESTING / BEHAVIOURAL TESTING

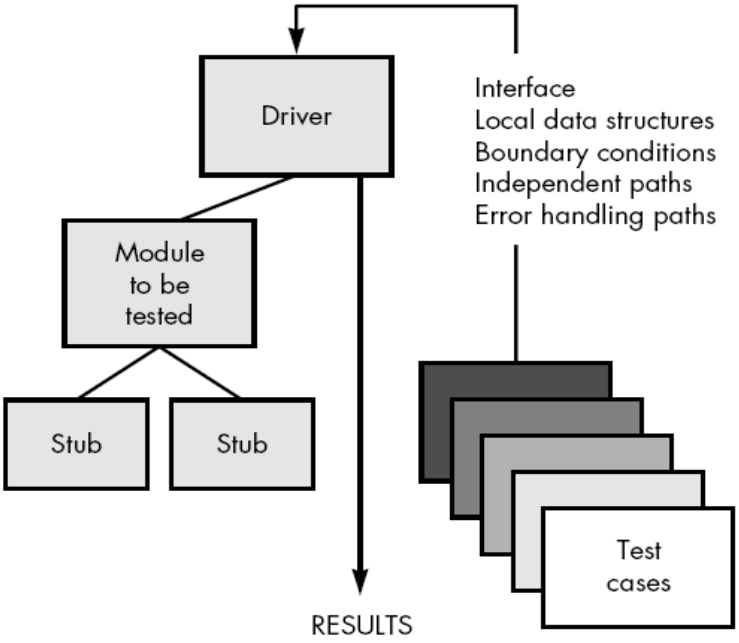
- Ensures the functionality of the system or program.
- Specification-based testing technique is also known as **'black-box'** or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.
- The testers have no knowledge of how the system or component is structured inside the box. In black-box testing the tester is concentrating on what the software does, not how it does it.
- The definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does its features or functions. Non-functional testing is concerned with examining how well the system does. Non-functional testing like performance, usability, portability, maintainability, etc.
- Specification-based techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. For example, when performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests.

- There are four specification-based or black-box technique:

    - Equivalence partitioning
    - Boundary value analysis
    - Decision tables
    - State transition testing

DRIVER AND STUB MODULES

- **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



- ✓ Stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.
- ✓ **For example:** Suppose you have a function (Function A) that calculates the total marks obtained by a student in a particular academic year. Suppose this function derives its values from another function (Function b) which calculates the marks obtained in a particular subject. You have finished working on Function A and wants to test it. But the problem you face here is that you can't seem to run the Function A without input from Function B; Function B is still under development. In this case, you create a dummy function to act in place of Function B to test your function. This dummy function gets called by another function. Such a dummy is called a Stub. To understand what a driver is, suppose you have finished Function B and is waiting for Function A to be developed. In this case you create a dummy to call the Function B. This dummy is called the driver.

EQUIVALENCE CLASS 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
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes – the two terms mean exactly the same thing.
- In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a

partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

GUIDELINES FOR DERIVING 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

- Boundary value analysis (BVA) is based on testing at the boundaries between partitions.
- Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).
- 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
- GUIDELINES FOR DERIVING EQUIVALENCE CLASSES

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*

2. If an input condition specifies a number of values, 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

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

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

- For example: An input field (Date of BIRTH) in which **Month** as an input data. Then the valid input should be any integer from 1 to 12, since there are only 12 months in a calendar year. So if the user enters the data as '25 ', then the system shouldn't accept it. For that we should write the program condition as (1<=month<=12)
- So in the above case, in **Equivalence class partitioning**, there are three classes. One valid class and two invalid class.
  Valid class: Any number in between 1 and 12 (1<=month<=12)
  Invalid class 1: Numbers less than 1 (Month<1)
  Invalid class 2: Numbers greater than 12 (Month>12)

So we should select a test case from each of the three equivalence classes. So the test suite may contain a number between 1 and 12, a number less than 1 and a number greater than 12.
Test suite = {6, -12, 30}

- So in the above case, in **Boundary value analysis**, the boundary values are 1 and 12. So choose a number just below the minimal condition value and choose a value just above the maximum condition value.
  So we should select the number just below 1 and just above 12 along with 1 and 12.
  Test Suite = {0,1,12,13}

WHITE BOX TESTING / STRUCTURAL TESTING / GLASS BOX TESTING

- Structure-based testing technique is also known as **'white-box'** or 'glass-box' testing technique because here the testers require knowledge of how the software is implemented, how it works.
- Focuses on the internal structure of the software or program.
- In white-box testing the tester is concentrating on how the software does it. For example, a structural technique may be concerned with exercising loops in the software.
- Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.
- Structure-based techniques can also be used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing, especially where there is good tool support for code coverage.
- Structure-based techniques are also used in system and acceptance testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.
- Uses two testing strategies: Fault based testing and Coverage based testing.

  - **Fault based testing:** It targets to detect certain types of faults. Mutation testing is a kind of fault based testing.
  - **Coverage based testing:** It attempts to execute (or cover) certain elements of a program. The various coverage based strategies are statement, branch, condition, path coverage-based testing.

STATEMENT COVERAGE

- Statement coverage based strategy aims to design test cases to ensure that , every statement is executed at least once.
- The statement coverage is also known as line coverage or segment coverage.
- The principal idea in statement coverage is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- The statement coverage **covers only the true conditions.**
- Through statement coverage we can identify the statements executed and where the code is not executed because of blockage.
- In this process each and every line of code needs to be checked and executed

Advantage of statement coverage:

- It verifies what the written code is expected to do and not to do
- It measures the quality of code written
- It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.

Disadvantage of statement coverage:

- It cannot test the false conditions.
- It does not report that whether the loop reaches its termination condition.
- It does not understand the logical operators.

The statement coverage can be calculated as shown below:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

BRANCH COVERAGE

- Decision coverage also known as branch coverage or all-edges coverage or edge testing.
- It **covers both the true and false conditions** unlikely the statement coverage.
- A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage
- A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF.

Advantages of decision coverage:

- To validate that all the branches in the code are reached
- To ensure that no branches lead to any abnormality of the program's operation
- It eliminate problems that occur with statement coverage testing

Disadvantages of decision coverage:

- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

The decision coverage can be calculated as given below:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

CONDITION COVERAGE

- This is closely related to decision coverage but has better sensitivity to the control flow.
- Test cases are designed to make each component of composite conditional expression to assume both true and false values.
- For a composite conditional expression with '**n**' components, there will be $\mathbf{2^n}$ test cases required to ensure condition coverage.

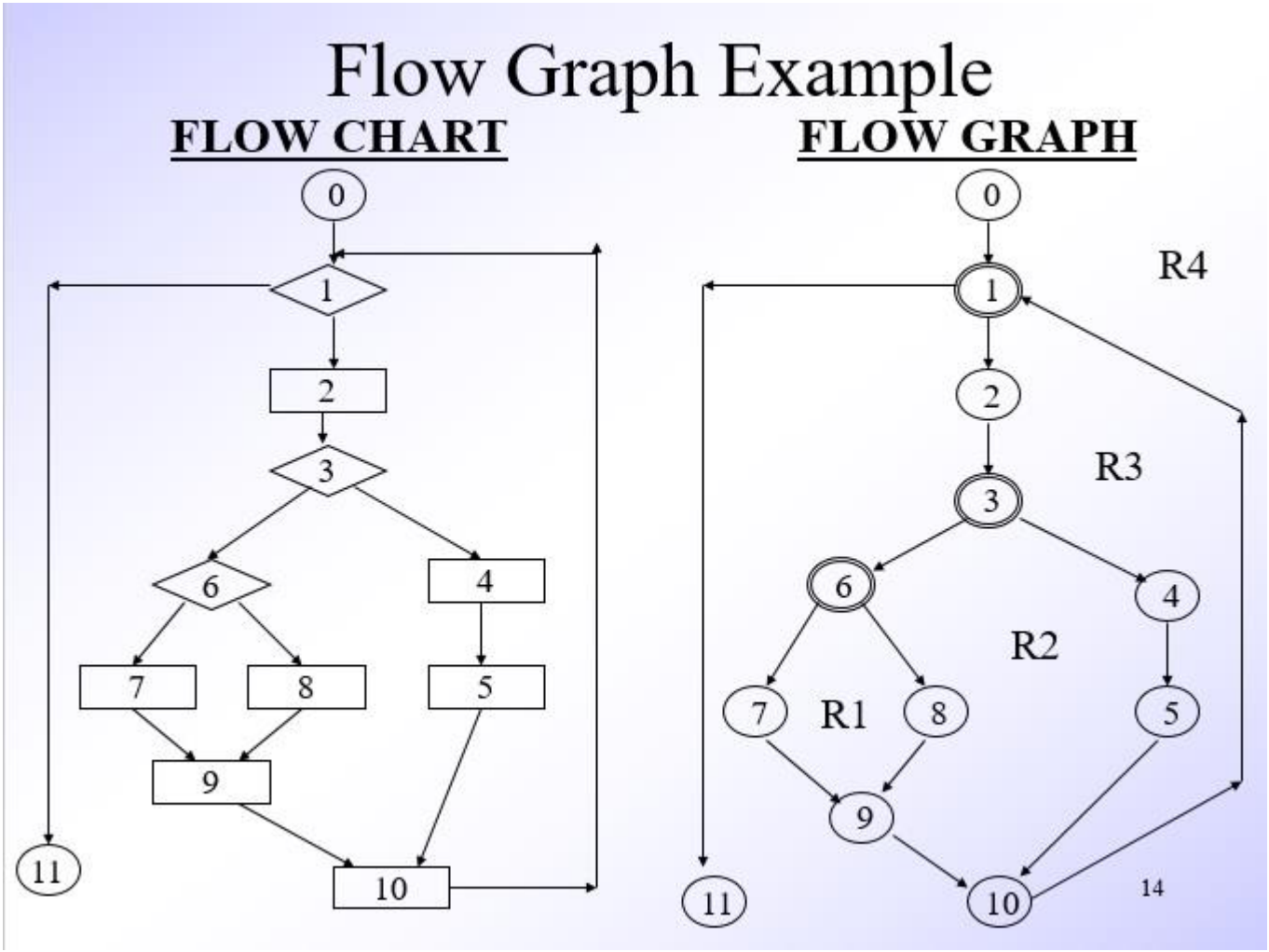
- However, full condition coverage does not guarantee full decision coverage.
- Condition coverage reports the true or false outcome of each condition.
- Condition coverage measures the conditions independently of each other.

PATH COVERAGE

- White-box testing technique proposed by Tom McCabe.
- It ensures that all linearly independent paths (or basis paths) in the program are executed at least once.
- 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
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

CFG 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



- 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 the above figure.
  - 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

❖ **MCCABE'S 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 / No. of closed regions + 1

- 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

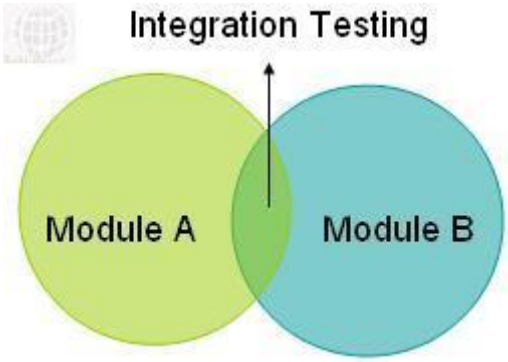
- Results in the following equations for the example flow graph shown in previous page.
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

❖ **STEPS FOR DERIVING BASIS SET & TEST CASES**

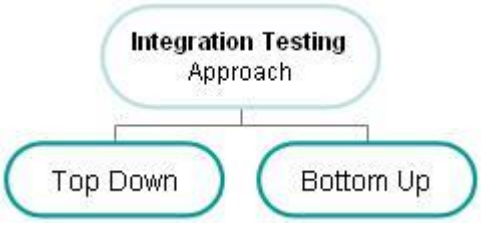
1. Using the design or code as a foundation, draw a corresponding flow graph
2. Determine the cyclomatic complexity of the resultant flow graph
3. Determine a basis set of linearly independent paths
4. Prepare test cases that will force execution of each path in the basis set

INTEGRATION TESTING

- Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.
- Also after integrating two different components together we do the integration testing. As displayed in the image below when two different modules 'Module A' and 'Module B' are integrated then the integration testing is done.



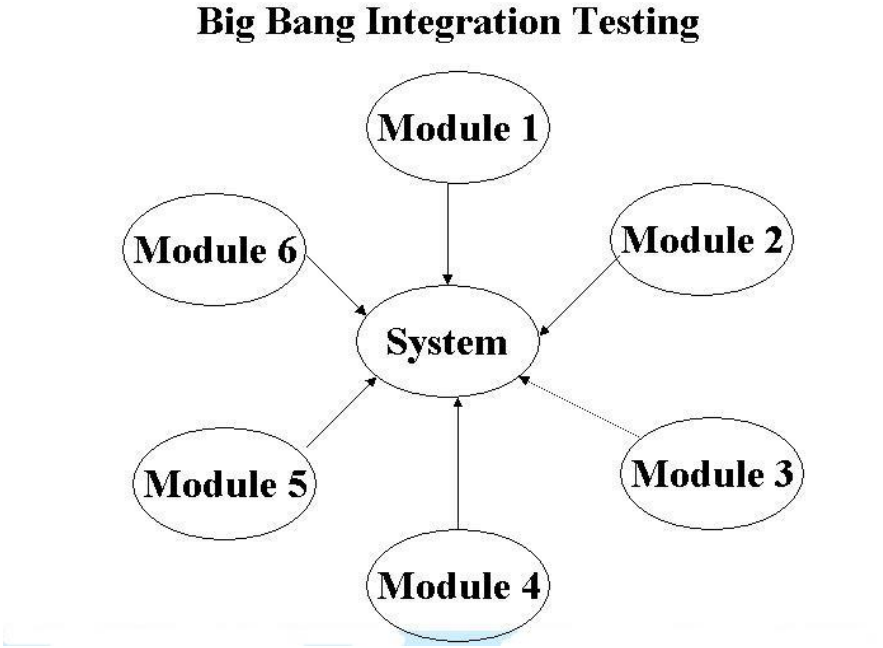
- Integration testing is done by a specific integration tester or test team.
- Integration testing follows two approach known as 'Top Down' approach and 'Bottom Up' approach as shown in the image below:



INTEGRATION TESTING TECHNIQUES

1. Big Bang integration testing:

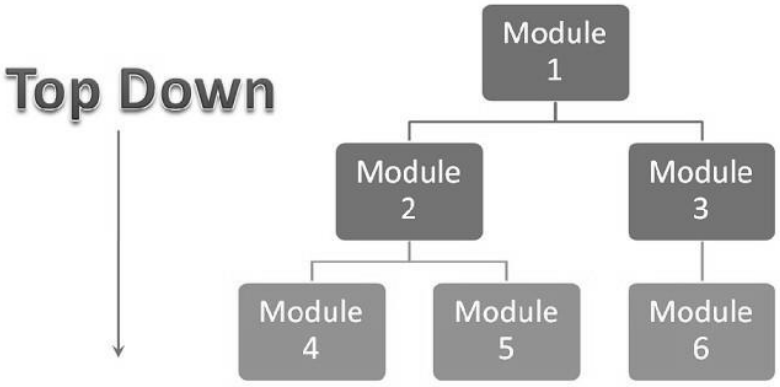
In Big Bang integration testing all components or modules are integrated simultaneously, after which everything is tested as a whole. As per the below image all the modules from 'Module 1′ to 'Module 6′ are integrated simultaneously then the testing is carried out.



Advantage: Big Bang testing has the advantage that everything is finished before integration testing starts.

Disadvantage: The major disadvantage is that in general it is time consuming and difficult to trace the cause of failures because of this late integration.

2. **Top-down integration testing:** Testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs. Below is the diagram of 'Top down Approach':



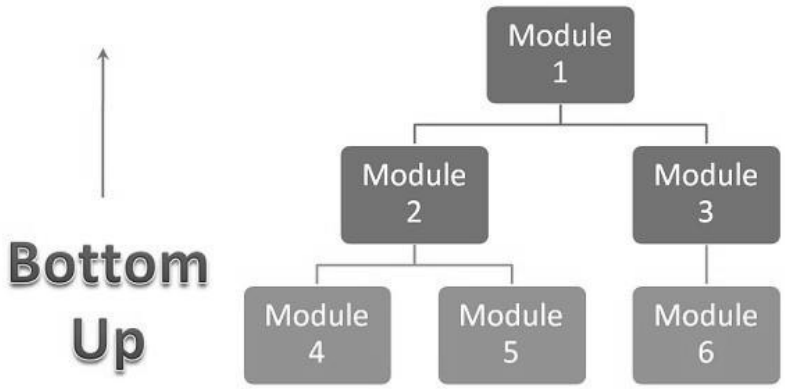
Advantages of Top-Down approach:

- The tested product is very consistent because the integration testing is basically performed in an environment that almost similar to that of reality
- Stubs can be written with lesser time because when compared to the drivers then Stubs are simpler to author.

Disadvantages of Top-Down approach:

- Basic functionality is tested at the end of cycle

3. **Bottom-up integration testing:** Testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers. Below is the image of 'Bottom up approach':



Advantage of Bottom-Up approach:

- In this approach development and testing can be done together so that the product or application will be efficient and as per the customer specifications.

Disadvantages of Bottom-Up approach:

- We can catch the Key interface defects at the end of cycle
- It is required to create the test drivers for modules at all levels except the top control

4. **Mixed / Sandwich Integration testing:** It follows the combination of both bottom up and top down testing approaches. In top down and bottom up testing, testing can be initiated only after the modules in the same level should be coded and unit tested. The mixed approach overcomes this shortcoming of top-down and bottom up approaches.

Advantage of Mixed integration:

- Testing can be started as and when modules become available after unit testing.

SYSTEM TESTING

- In system testing the behaviour of whole system/product is tested as defined by the scope of the development project or product.
- It may include tests based on risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behaviour, interactions with the operating systems, and system resources.
- System testing is most often the final test to verify that the system to be delivered meets the specification and its purpose.
- System testing is carried out by specialist testers or independent testers.
- System testing should investigate both functional and non-functional requirements of the testing.

ALPHA TESTING

- System testing done by the developer itself. This test takes place at the developer's site.
- Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.
- Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.
- Alpha testing is final testing before the software is released to the general public. It has two phases:
    - In the **first phase** of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.
    - In the **second phase** of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

- Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

BETA TESTING

- Testing done by a friendly set of customers.
- It is also known as field testing. It takes place at **customer's site**. It sends the system to users who install it and use it under real-world working conditions.
- A beta test is the second phase of software testing in which a sampling of the intended audience tries the product out. (Beta is the second letter of the Greek alphabet.) Originally, the term *alpha test* meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing. Beta testing can be considered "pre-release testing.
- The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final, released version of the application.

ACCEPTANCE TESTING

- Testing done by the customer to decide whether to accept or reject the product.
- After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing.
- Acceptance testing is basically done by the user or customer although other stakeholders may be involved as well.

- The goal of acceptance testing is to establish confidence in the system.
- Acceptance testing is most often focused on a validation type testing.

REGRESSION TESTING

- ☐ During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these '**unexpected side-effects**' of fixes is to do regression testing.
- ☐ The purpose of a regression testing is to verify that modifications in the software or the environment have not caused any unintended adverse side effects and that the system still meets its requirements.
- ☐ Regression testing are mostly automated because in order to fix the defect the same test is carried out again and again and it will be very tedious to do it manually.
- ☐ Regression tests are executed whenever the software changes, either as a result of fixes or new or changed functionality.

DEBUGGING

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

BRUTE FORCE:

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

BACKTRACKING

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

CAUSE ELIMINATION

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- 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
- Alternatively, 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

PROGRAM SLICING

- Similar to back tracking, but the search space is reduced by defining slices.
- A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

TEST PLAN

- ☐ Test plan is the project plan for the testing work to be done. It is not a test design **specification,** a collection of **test cases** or a set of **test procedures;** in fact, most of our test plans do not address that level of detail.
- ☐ A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.
  Master test plan: A test plan that typically addresses multiple test levels.
  Phase test plan: A test plan that typically addresses one test phase.

TEST PLAN GUIDELINES

- Make the plan concise. Avoid redundancy in your test plan.
- Be specific. For example, when you specify an operating system as a property of a test environment, mention the OS Edition/Version as well, not just the OS Name.
- Make use of lists and tables wherever possible. Avoid lengthy paragraphs.
- Have the test plan reviewed a number of times prior to baselining it or sending it for approval. The quality of your test plan speaks volumes about the quality of the testing you or your team is going to perform.
- Update the plan as and when necessary. An out-dated and unused document stinks and is worse than not having the document in the first place.

IEEE 829 STANDARD TEST PLAN TEMPLATE

Test plan identifier
Test deliverables
Introduction
Test tasks
Test items
Environmental needs
Features to be tested
Responsibilities
Features not to be tested
Staffing and training needs
Approach Schedule
Item pass/fail criteria
Risks and contingencies
Suspension and resumption criteria Approvals

TEST REPORTING

Test completion reporting is a process where test metrics are reported in summarised format to update the stakeholders which enables them to take an informed decision.

Test Completion Report Format:

- Test Summary Report Identifier
- Summary
- Variances
- Summary Results
- Evaluation
- Planned vs Actual Efforts
- Sign off

Significance of Test Completion Report:

- An indication of the quality
- Measure outstanding risks
- The level of confidence in tested software

SAMPLE TEST REPORT

TEST CASE REPORT

(Use one template for each test case)

GENERAL INFORMATION			
Test Stage:	☐ Unit ☐ Functionality ☐ Integration ☐ System ☐ Interface ☐ Performance ☐ Regression ☐ Acceptance ☐ Pilot Specify the testing stage for this test case.		
Test Date:	mm/dd/yy	System Date, if applicable:	mm/dd/yy
Tester:	Specify the name(s) of who is testing this case scenario.	Test Case Number:	Specify a unique test number assigned to the test case.
Test Case Description:	Provide a brief description of what functionality the case will test.		
Results:	☐ Pass ☐ Fail	Incident Number, if applicable:	Specify the unique identifier assigned to the incident.
INTRODUCTION			
Requirement(s) to be tested:	Identify the requirements to be tested and include the requirement number and description from the Requirements Traceability Matrix.		
Roles and Responsibilities:	Describe each project team member and stakeholder involved in the test, and identify their associated responsibility for ensuring the test is executed appropriately.		
Set Up Procedures:	Describe the sequence of actions necessary to prepare for execution of the test.		
Stop Procedures:	Describe the sequence of actions necessary to terminate the test.		
ENVIRONMENTAL NEEDS			
Hardware:	Identify the qualities and configurations of the hardware required to execute the test case.		