Testing:

- Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not.
- Testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.
- Testing can be defined as - A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.
- New systems are computerized; therefore it is important for new systems to be implemented using a conversion strategy.

Purpose of 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.
- To improve quality
- For verification and validation (V&V)
- For reliability estimation

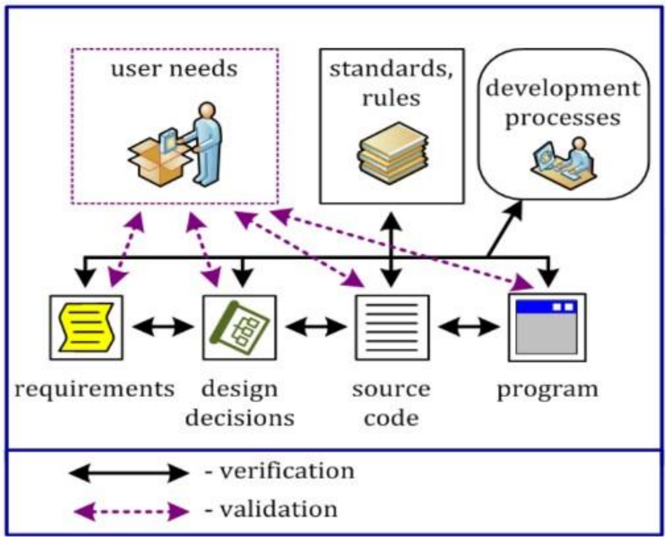
Testing Terminology:

1. Verification:

- Verification is the process, to ensure that whether we are building the product right i.e., to verify the requirements which we have and to verify whether we are developing the product accordingly or not.
- Activities involved here are Inspections, Reviews and Walkthrough.
- It is a static testing method.

2. Validation:

- Validation is the process, whether we are building the right product i.e., to validate the product which we have developed is right or not.
- Activities involved in this is Testing the software application.
- It is a dynamic testing method.



3. Debugging:

- Debugging, in computer programming and engineering, is a multistep process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it.
- The final step of debugging is to test the correction or workaround and make sure it works.

Testing Pieces:

1. Test case:

- Test cases describe a specific idea that is to be tested, without detailing the exact steps to be taken or data to be used.
- For example, a test case might say “Test that discount codes can be applied on top of a sale price.” This doesn’t mention how to apply the code or whether there are multiple ways to apply the code. The actual testing that will cover this test case may vary from time to time.
- Will the tester use a link to apply a discount, or enter a code, or have a customer service representative apply the discount, or will they feel compelled to test every way to add a discount that they can think of?
- Test cases give flexibility to the tester to decide exactly how they want to complete the test.
- This flexibility from test cases is both good and bad. Flexibility is beneficial when the tester is familiar with testing and familiar with the software under test and the current set of risks in the software. If the tester clearly understands what has already been tested, what has changed recently in the program, and how users typically use the program, they will choose an approach in their testing that will exercise both the most common user paths, and the less common paths that are most likely to reveal bugs.
- On the other hand, if the tester does not have a good understanding of how the program is used, the recent risks to the program, and how to evaluate those risks as a tester, they may not have the information or skill they need to assess the actions required to reveal important bugs.

2. Test scripts:

- When people talk about test scripts, they usually mean a line-by-line description of all the actions and data needed to perform a test. A script typically has ‘steps’ that try to fully describe how to use the program —

which buttons to press, and in which order — to carry out a particular action in the program. These scripts also include specific results that are expected for each step, such as observing a change in the UI. An example step might be “Click the ‘X’ button,” with an example result of “The window closes.”

- When a tester first starts a new job, they might not know much about the product, the business domain, or even software testing. Scripts can help bridge that gap. If the tester carefully follows the directions — enter the string ‘abc’, click the submit button, make sure the form submitted and the value was saved — the test idea will be covered enough to consider it ‘tested’.
- There are a few drawbacks to consider before going all-in with detailed scripts:
    - Active software projects change often — pages get redesigned, user experience changes, and new functionality is added. To be effective over time, testers have to make a continuous effort to update the scripts to match the new product. This can take time away from testing.
    - Scripted tests are often designed to test one specific thing repeatedly, using the same steps and the same data each time the test is executed. This means that if there are bugs that lie outside the directions given in the test script, they will not be found unless the tester strays from the script. Scripted tests do not always encourage testers to use the creativity and technical skill required to find hidden bugs.

3. Test scenario:

- The least detailed type of documentation is the test scenario.
- A test scenario is a description of an objective a user might face when using the program.

- An example might be “Test that the user can successfully log out by closing the program.” Typically, a test scenario will require testing in a few different ways to ensure the scenario has been satisfactorily covered.
- Just based on that light description, the tester might choose to close the program through the menu option, kill it through the task manager, turn the computer off, or see what happens when the program runs out of memory and crashes. Since test scenarios offer little information about how to complete the testing, they offer the maximum amount of flexibility to the tester responsible for them.

4. Test plan:

- A Software Test Plan is a document describing the testing scope and activities. It is the basis for formally testing any software/product in a project.
- 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.

5. Test harness:

- Test harness enables the automation of tests. It refers to the system test drivers and other supporting tools that are required for executing tests. It provides stubs and drivers which are small programs that interact with the software under test.
- Test harness executes tests, by using a test library and generates a report. It requires that your test scripts are designed to handle different test scenarios and test data.

6. Test suits:

- A test suite, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors.
- A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the next tests.

Testing principles:

- All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer’s point of view) are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. All tests can be planned and designed before any code has been generated.
- The Pareto principle applies to software testing. Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate [separate] these suspect components and to thoroughly test them. Eg. 20 percent of software bugs cause 80 percent of the software’s failures.
- Testing should begin “in the small” and progress toward testing “in the large.” The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
- Exhaustive testing is not possible. The number of path permutations [combinations] for even a moderately sized program is exceptionally

large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately [satisfactorily] cover program logic and to ensure that all conditions in the component-level design have been exercised.

- To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing).

<u>**White box testing**</u>:

- White-box testing is the detailed investigation of internal logic and structure of the code.
- The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.
- WHITE BOX TESTING is also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing.
- It is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
- The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.
- This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.
- **Advantages:**
    - Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

- Testing is more thorough, with the possibility of covering most paths.
- As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.
- It helps in optimizing the code.
- Extra lines of code can be removed which can bring in hidden defects.
- Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.
- Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.

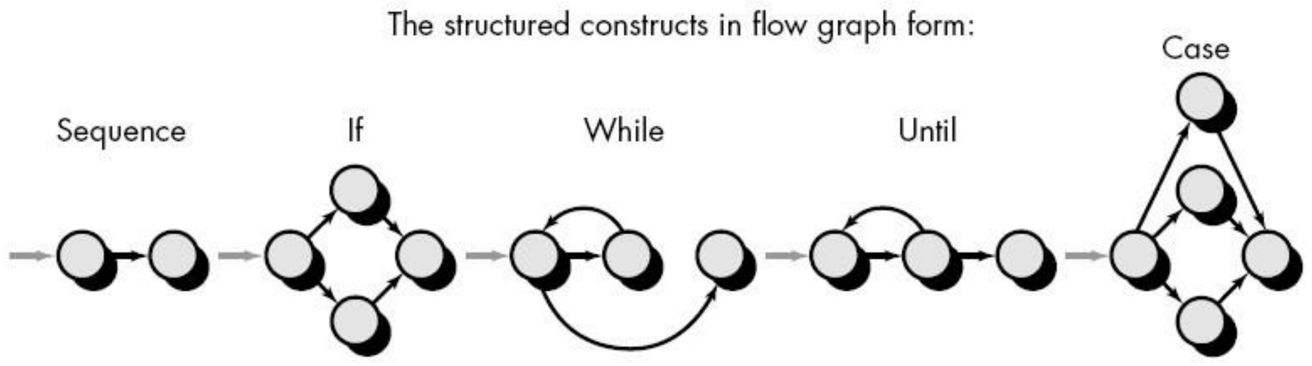
- **Disadvantages:**
  - Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
  - It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.

<u>**Methods of White Box Testing**</u>:

1. Basis Path Testing:

- **Flow graph notation:**
  - Referring to the figure, each circle, called a flow graph node, represents 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, represent flow of control. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).

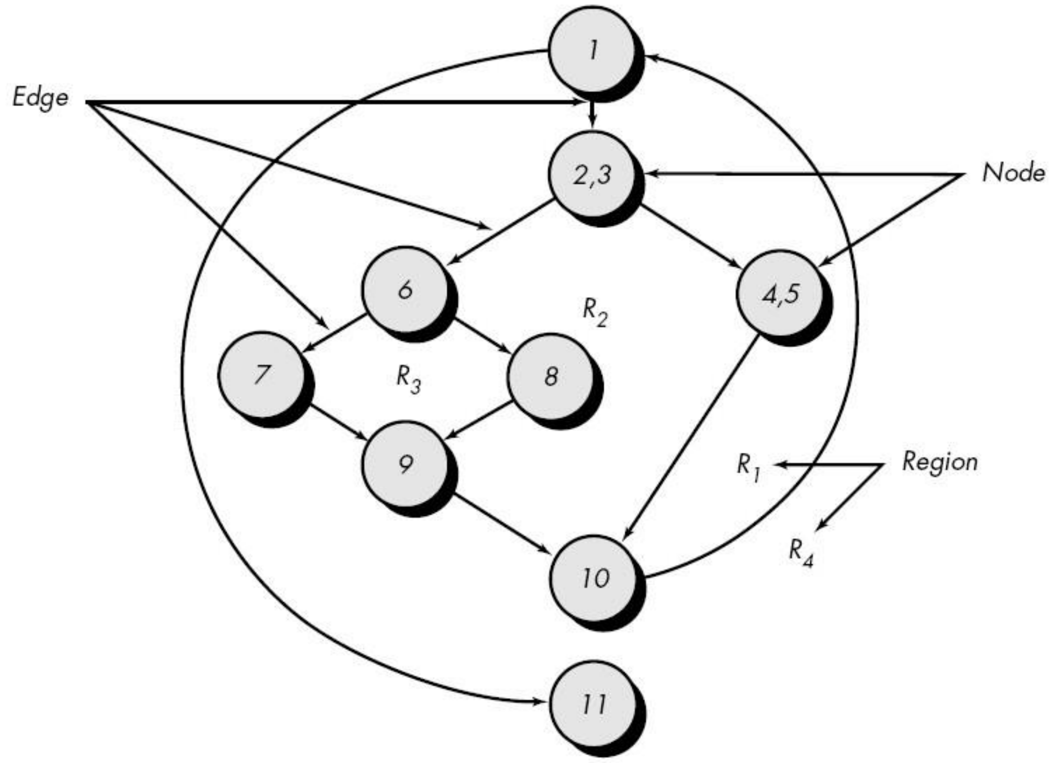
- Areas bounded by edges and nodes are called **regions**. When counting regions, we include the area outside the graph as a region. Each node that contains a condition is called a **predicate nodes** and is characterized by two or more edges emanating from it.



- **Independent program path:**
  - An independent program path is any program that introduce at least one new set of processing statement or a new condition.
  - For example: paths illustrated in figure
    - Path 1: 1 – 11
    - Path 2: 1 – 2 – 3 – 4 – 5 – 10 – 1 - 11
    - Path 3: 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 - 11
    - Path 4: 1- 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11
  - How many paths to look for can be counted by using cyclomatic complexity?
  - **Cyclomatic complexity** is a software metric that provides a quantitative measure of the logical complexity of a program.
  - Three ways to identify cyclomatic complexity:
    - The number of regions
    - $V(G)=E - N + 2$

      → Where, E= number of edges in a flow graph and N = Number of nodes in a flow graph
    - $V(G)=P + 1$

→ Where, P= Predicate nodes in a flow graph



- **Deriving test cases:**
    - Steps to derive test cases:
        - Draw a corresponding flow graph
        - Determine cyclomatic complexity
        - The number of regions Eg. 4

          V(G)=E – N + 2 Eg : V(G)=11 – 9 + 2 =4

          V(G)=P + 1 Eg. V(G) = 3 +1 =4
        - Determine a basis set of linearly independent paths

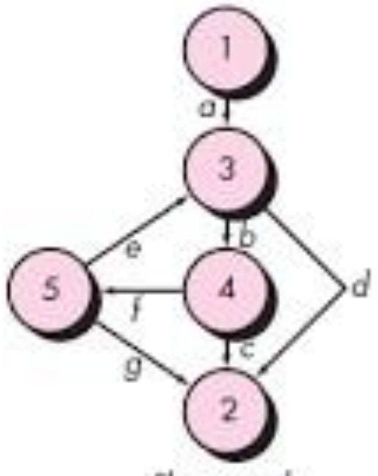
          Path 1: 1 – 11

          Path 2: 1 – 2 – 3 – 4 – 5 – 10 – 1-...

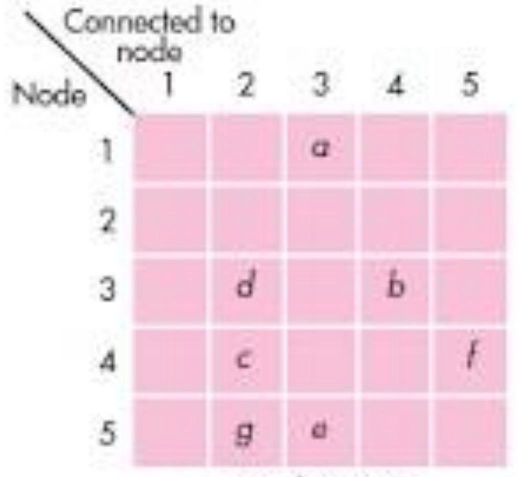
          Path 3: 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1-...

          Path 4: 1- 2 – 3 – 6 – 7 – 9 – 10 – 1 -...
        - (...) indicates that any path through the remainder of the control structure is acceptable.

- **Graph matrices:**
  - In graph matrix based testing, we convert our flow graph into a square matrix with one row and one column for every node in the graph. If the size of graph increases, it becomes difficult to do path tracing manually.
  - A graph matrix is a square matrix whose size is equal to the number of nodes on the flow graph. It is the tabular representation of a flow graph and use to develop tool that assist in basis path testing.



Flow graph



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

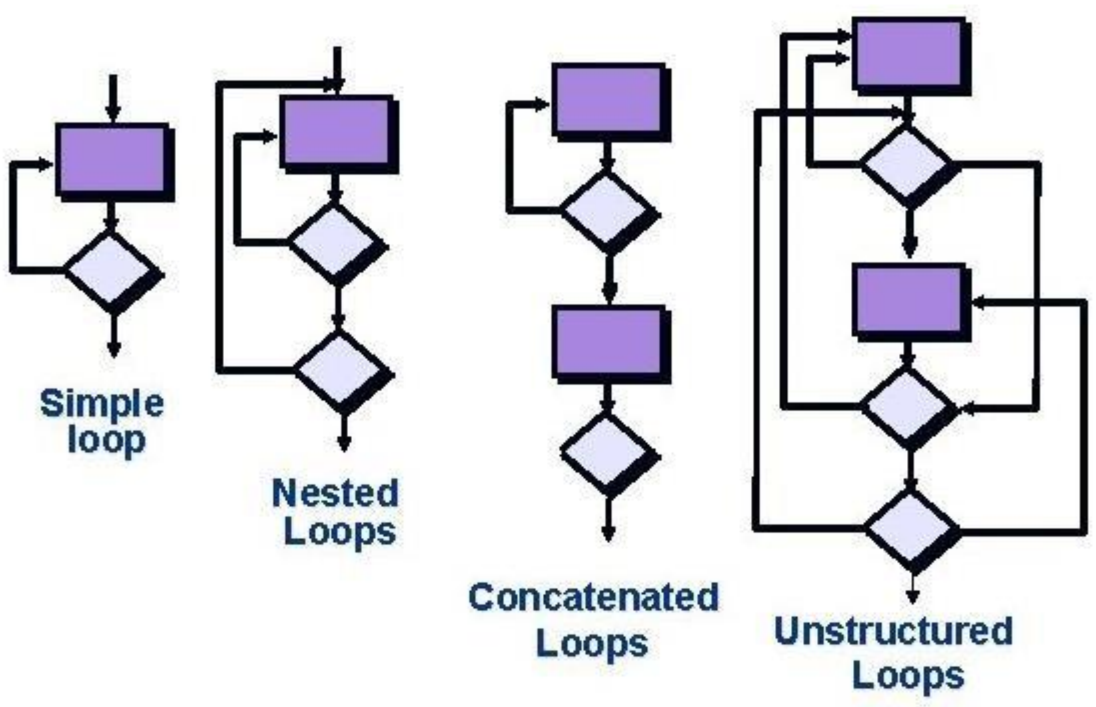
Graph matrix

2. Control structure testing:

- **Conditions Testing:**
  - Condition testing aims to exercise all logical conditions in a program module. Logical conditions may be complex or simple. Logical conditions may be nested with many relational operations.
  - Relational expression: (E1 op E2), where E1 and E2 are arithmetic expressions. For example, (x+y) – (s/t), where x, y, s and t are variables.
  - Simple condition: Boolean variable or relational expression, possibly proceeded by a NOT operator.
  - Compound condition: composed of two or more simple conditions, Boolean operators and parentheses along with relational operators.

- Boolean expression: Condition without relational expressions

- **Loop Testing:**
  - Loops are fundamental to many algorithms. Loops can be categorized as, define loops as simple, concatenated, nested, and unstructured.



  - Loops can be defined in many ways.
  - **Testing of Simple Loops**
    - Skip the loop entirely
    - Only one pass through the loop
    - Two passes through the loop
    - m passes through the loop, where m < n
    - n –1, n, n + 1 passes through the loop
    - ‘n’ is the maximum number of allowable passes through the loop
  - **Testing of Nested Loops**
    - Start at the innermost loop; set all other loops to minimum values

    - 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
    - Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
    - Continue until all loops have been tested
  - **Testing of Concatenated Loops**
    - For independent loops, use the same approach as for simple loops
    - Otherwise, use the approach applied for nested loops
  - **Testing 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

<u>Black box testing</u>:

- The technique of testing without having any knowledge of the interior workings of the application is called black-box testing. .
- The tester is oblivious to the system architecture and does not have access to the source code.
- Typically, while performing a black-box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.
- It is also known as closed box, behavioral testing and functional testing.
- This type testing is designed to give answers the following types of questions:
  - How is functional validity tested?
  - How are system behavior and performance tested?

    - What classes of input will make good test cases?
    - Is the system particularly sensitive to certain input values?
    - How are the boundary values of a data class isolated?
    - What data rates and data volume can the system tolerate?
    - What effect will specific combinations of data have on system operation?
- It attempts to find the errors of following categories:
    - Incorrect or missing functions
    - Interface errors
    - Errors in data structures or external data base access
    - Behavior or performance errors
    - Initialization and termination errors
- **Advantages:**
    - Well suited and efficient for large code segments.
    - Code access is not required.
    - Clearly separates user's perspective from the developer's perspective through visibly defined roles.
    - Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.
    - Limited coverage, since only a selected number of test scenarios is actually performed.
- **Disadvantages:**
    - Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
    - Blind coverage, since the tester cannot target specific code segments or error-prone areas.
    - The test cases are difficult to design.

<u>Methods of Black Box Testing</u>:

1. 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.
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once.
- If an input condition specifies
- **Guidelines for Defining Equivalence Classes:**
    - If an input condition specifies <u>a range</u>, one valid and two invalid equivalence classes are defined
      Input range: 1 – 10 Eg classes: {1..10}, {x < 1}, {x > 10}
    - If an input condition requires <u>a specific value</u>, one valid and two invalid equivalence classes are defined
      Input value: 250 Eg classes: {250}, {x < 250}, {x > 250}
    - If an input condition specifies a member of <u>a set</u>, one valid and one invalid equivalence class are defined
      Input set: {-2.5, 7.3, 8.4} Eg classes: {-2.5, 7.3, 8.4}, {any other x}
    - If an input condition is <u>a Boolean value</u>, one valid and one invalid class are define
      Input: {true condition} Eg classes: {true condition}, {false condition}

2. 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
- **Guidelines for Defining Boundary Value Analysis:**
    - 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
    - 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

Unit testing:

- This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases.
- Unit testing is performed by the respective developers on the individual units of source code assigned areas.
- The developers use test data that is different from the test data of the quality assurance team.

- The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Integration testing:

- Integration testing is defined as the testing of combined parts of an application to determine if they function correctly.
- Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing.
- Integration testing can be done in two ways:
- **Bottom-up integration:**
  - This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
- **Top-down integration:**
  - In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

System testing:

- System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards.
- This type of testing is performed by a specialized testing team.
- **System testing is important because of the following reasons:**
  - System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
  - The application is tested thoroughly to verify that it meets the functional and technical specifications.

- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

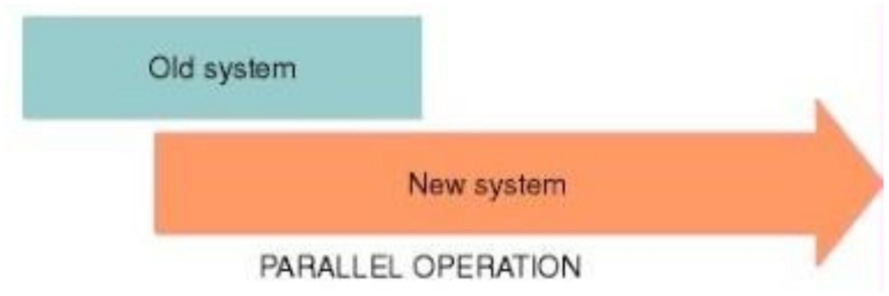
Changeover:

- A system changeover is **the process of transitioning from one system to another**.
- This can be a major or minor change, and can involve changes to personnel, processes, or technology.
- Conversion is the process of switching from something old to new, so, system conversion is the process of switching from an old system to a new system.
- The reason why system conversions need to take place is because existing systems may be outdated or they may not cater for the users needs any more.

Types of changeover:

1. Parallel System Method:

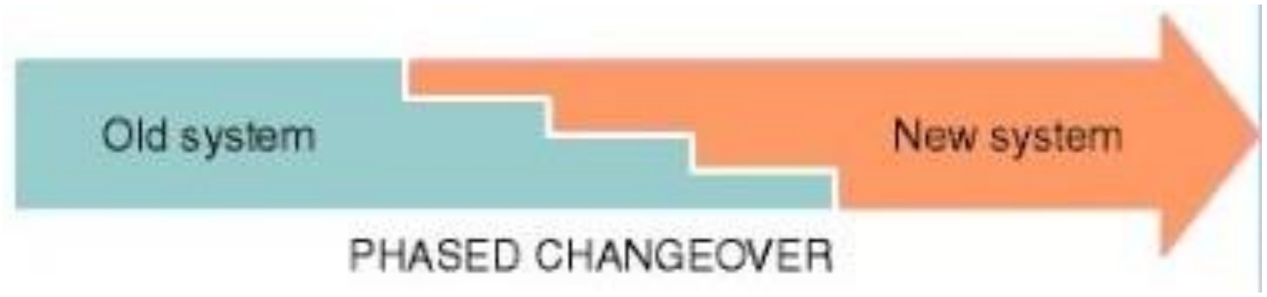
- The parallel conversion strategy is when both the old and new information systems operate alongside one another for a specified time.
- Once the results have been compared between the old and new system, the organization may choose to gradually welcome in the new system or immediately end the previous system.
- Of all the techniques, this tends to be the most popular, mainly because it carries the lowest risk. If something goes wrong at any point, the entire system can be reverted back to its original state.



- **Advantages:**
  - By using the parallel method, small minor errors can be easily seen
  - Companies are able to fix any problems with the new system before ending the previous system
- **Disadvantages:**
  - It is very costly as two systems are being operated simultaneously, so there will be the costs for more power for example
  - Operating two systems simultaneously is also very time consuming and stressful as there is more work involved, such as creating more reports

2. Dual System Method or Phase-in Method:

- In a phased changeover, the new system is implemented one stage at a time.
- As an example, consider a company working toward installing a new financial system. Implementing the new system one department at a time, the company converts accounts receivable, accounts payable, payroll, and so on.



- **Advantages:**

- As the system is tested at every stage, there is very little chance of error
- This strategy is more user friendly. Because the new system is implemented one department at a time, the IT staff is able to draw their attention to training one department effectively to using the new system.

➢ **Disadvantages:**

- It takes a lot of time to implement the whole new system to the entire organization.

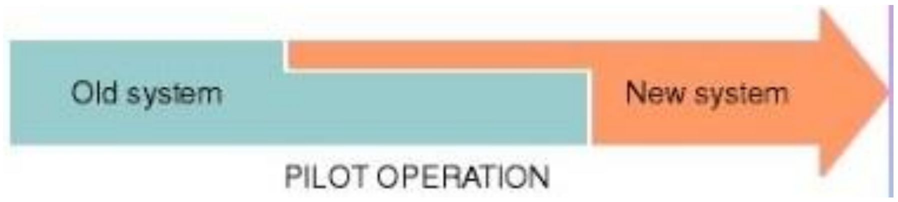
3. Direct Method:

➢ Direct changeover, also referred to as immediate replacement, tends to be the least favorite of the changeover techniques.

➢ In a direct changeover, the entire system is replaced in an instant. Basically, as soon as the new system is powered up, the old system is shut down.

➢ This type of changeover carries the most risk because; if something goes wrong, reverting back to the old system usually is impossible.

➢ Using the direct changeover technique tends to work best in situations where a system failure isn't critical enough to result in a disaster for the company.



➢ **Advantages:**

- It is less costly as it is a direct change over
- It is not very time consuming as once the old system has stopped being used the new system is immediately being set up

- **Disadvantages:**
  - If the system has not been implemented properly the new system may fail to work and this will affect the whole organization.
  - It is very difficult to detect small errors in the new system

4. Pilot Approach:

- With a pilot changeover, the new system is tried out at a test site before launching its company-wide.
- This strategy is mainly used for testing the new system in different environments.
- For example, a bank may first test the system at one of its branches. This branch is referred to as the pilot, or beta, site for the program.
- Using the pilot changeover technique allows companies to run the new system next to their old but on a much smaller scale.
- This makes the pilot changeover method much more cost-effective.
- After the kinks are worked out of the system at the test site, companies usually opt to use the direct changeover technique to launch the system company-wide.

Old system
New system
DIRECT CUTOVER

- **Advantages:**
  - Risk is reduced
  - Allows the organization to see whether the new system will meet the organizations needs in one department/location before using it throughout the entire organization.
- **Disadvantages:**
  - Too much time is involved testing in one location, there is also increased development and labor costs.