ECHLEON INSTITUTE OF TECHNOLOGY

UNIT 2

SOFTWARE TESTING

TESTING STRATEGIES

What is White Box Testing in SDLC?

A software testing method known as "White Box Testing" focuses on looking at the internal workings and implementation specifics of the software under test. It is also known as structural testing, clear box testing, and transparent box testing. In white box testing, the tester has access to the software system's internal code, algorithms, and design. As a result, they can produce test cases that focus on certain program logic, conditions, and paths.

Software developers or specialized testers who have access to the source code often carry out white-box testing. It complements other testing strategies like black box testing, in which the tester is unaware of how the software internally functions. Software engineers can improve the overall quality, dependability, and security of the software system by combining several testing techniques.

What are the Objectives of White Box Testing?

White box testing's primary goals are to make sure of the following:

1. **Confirm the Accuracy of Internal Operations:** White box testing looks at the internal logic, flow, and behavior of the code to confirm that the product works as intended. Designing test cases to disclose potential flaws, faults, or vulnerabilities requires analyzing the code to find them.
2. **Achieve the Highest Possible Code Coverage:** White box testing seeks to achieve thorough code coverage by putting the code's many pathways, statements, and branches to the test. Testing not only the program's main flow but also diverse edge cases and extraordinary scenarios is the aim.
3. **Boost Software Performance:** White box testing can locate areas that might have an impact on performance by identifying the underlying organization and algorithms of the software. Testers can assess the code's effectiveness and resource utilization and make suggestions for performance enhancements.
4. **Validate Security and Robustness:** White box testing can assist in locating software flaws and security holes. Testers can identify potential security holes and recommend solutions to strengthen the system's resistance to attacks by carefully reviewing the code and how it interacts.

What Is the Focus of White Box Testing?

White box testing uses in-depth internal application knowledge to create test scenarios that are extremely focused. White box testing may involve the following types of tests, for instance:

1. **Path checking:** To check that all conditional statements are accurate, required, and effective, white box testing can be used to investigate the many execution routes within an application.
2. **Output Validation:** A function's potential inputs are listed in the output validation process, which checks that each one leads to the desired outcome.
3. **Security testing:** To find potential flaws in an application and confirm that it adheres to secure development best practices, static code analysis and other white box testing techniques are utilized.
4. **Loop testing:** Loop testing verifies that an application's loops are accurate, effective, and effectively manage the variables under their purview.
5. **Data Flow Testing:** Verifies that variables are declared, initialized, utilized, and correctly manipulated by following variables along a program's execution routes.

Types of White Box Testing in SDLC

White box testing can be used for a variety of reasons. There are three varieties of white box testing:

1. **Unit Testing**: Unit testing is used to validate the functionality of each component or feature of an application. This makes it easier to make sure that during the development phase, the application complies with design criteria.
2. **Integration testing**: Integration testing is concerned with how different parts of an application interact with one another. Following unit testing, it makes sure that each component not only functions well on its own but also that it can cooperate with other components to produce the desired result.
3. **Regression testing**: Modifications have the potential to break an application. Regression testing ensures that the code still passes the earlier test cases after an application receives functionality or security updates.

What to Verify in White Box Testing?

The following aspects of the software code are tested during white box testing:

- Intra-company security gaps
- Paths in the coding processes that are broken or poorly organized
- The way particular inputs are passed through the code
- Anticipated result
- Conditional loops' capabilities
- Each statement, object, and function is independently tested

The system, integration, and unit testing are all possible during the software development process. Verifying that an application's working flow is one of white box testing's fundamental objectives. It is comparing a sequence of specified inputs to desired or expected outputs in order to identify bugs when a particular input does not provide the desired outcome.

How is White Box Testing Carried Out?

We've divided white box testing into two straightforward stages to make it simpler for you to comprehend. Using the white box testing approach, testers do the following tasks when assessing an application:

1. **Understand the source code:** A tester will frequently study and comprehend the application's source code as their initial step. White box testing entails checking an application's internal workings; therefore, the tester must have a thorough understanding of the programming languages used in the apps they are assessing. Additionally, the tester needs to be well-versed in secure coding practices. Often, one of the main goals of software testing is security. The tester should be able to identify security flaws and thwart assaults from hackers and gullible users who might willfully or accidentally introduce malicious code into the application.
2. **Creation of test cases and execution:** The second key step in white box testing comprises checking the application's source code to make sure it is organized and flows properly. Writing more code to test the source code of the application is one approach. For each procedure in the application—or set of processes—the tester will create a little test. This method, which is frequently carried out by the developer, necessitates that the tester has a thorough understanding of the code. As we shall discuss later in this article, other techniques include manual testing, trial-and-error testing, and the use of testing tools.

What are the Different Techniques Used in White Box Testing?

The following are some examples of white box testing techniques:

Statement Coverage:

With this technique, every single statement in the code is run at least once. To guarantee that each line of code is run during testing, test cases are created. A white-box testing technique called "statement coverage" seeks to make sure that each statement in a program is run at least once during

testing. It calculates the proportion of the code that has been run through the test cases. Consider a straightforward example in order to clarify the idea. Suppose we have a function in a programming language called calculated that computes the addition of two numbers and returns the result:

```
def calculateSum(a, b):
if a > 0 and b > 0:
     result = a + b
    print("The sum is:", result)
else:
    print("Invalid input!")
```

We want to write test cases for every statement in this code, which is known as statement coverage. Consider the following test cases:

Test Case 1: calculateSum(2, 3)
Test Case 2: calculateSum(-1, 5)
Test Case 3: calculateSum(0, 0)
Test Case 4: calculateSum(4, 0)
Let's now examine how well these test scenarios are covered:

- All of the statements in the code are run in Test Case 1. Within the if block, the print statement and the if condition are both executed.
- The else block in Test Case 2 is executed, but the if block is not. Therefore, the coverage is lacking.
- Neither the if block nor the else block is executed in Test Case 3. It doesn't cover any of the code's statements.
- The if block in Test Case 4 is executed, but the else block is not. There is insufficient coverage.

The test cases in this sample covered the following ground:
75% of statements were performed (3 of 4 statements). We would need to create more test cases that cover the remaining statements in order to achieve 100% statement coverage. For instance, we may develop a test case that runs the code's else block. Aiming for high statement coverage improves the chance of finding program faults or probable errors. It's crucial to remember, though, that achieving high statement coverage simply assesses the thoroughness of statement execution and does not ensure the absence of defects in the code. Branch coverage and path coverage are two further white-box testing methods that can give more thorough coverage and identify more problems.

Branch Coverage:

This method makes sure that both the true and false branches of conditional statements are performed by testing every potential decision outcome. Branch coverage is a white-box testing approach used to assess how thoroughly a program's branches have been tested. It seeks to make sure that every decision point or branch that could possibly exist in the code has undergone at least one test. Testers can feel confident in the software's dependability and accuracy by achieving high branch coverage. Every decision point in the code is taken into account for branch coverage. An if statement, switch statement, or loop are common examples of decision points in code, which allow the program to choose between two or more options. Exercises are intended for both true and false decision-point outcomes.
Consider a straightforward example to show branch coverage. Let's say we have a function that determines the highest value among two input parameters, x, and y:

```
function findMax(x, y):
if x > y:
```

```
    maxVal = x
else:
    maxVal = y
return maxVal
```

We would need to create test cases that cover both the true and false outcomes of the decision point at line 2 in order to achieve 100% branch coverage. We could, for instance, develop the following test cases:

Test Case 1:

x = 5 y = 3

output anticipated: maxVal = 5 (True: Branch taken)

Test Case 2:

x = 2 y = 7

output anticipated: maxVal = 7 False (Branch taken)

We would exercise both branches (true and false) of the decision point by running these two test cases, completing full branch coverage.

Path Coverage:

A white box testing technique called path coverage makes sure that all feasible routes through a program or system are tested in order to assess how to complete test cases are. By testing each potential path from the beginning to the finish of a function, method, or program, it seeks to achieve maximum code coverage. All potential software pathways, including loops, conditionals, and nested structures, are tested as part of path coverage. This method makes certain that each potential execution path is examined.

Let's explore an example to better grasp path coverage. Let's say we have a straightforward function called "calculateSum" that takes two integer inputs and outputs their sum:

```
function calculateSum(a, b):
if a > 0:
    result = a + b
    print("Sum is positive")
else:
    result = a – b
    print("Sum is negative")
return result
```

According to whether the sum of the two integers is positive or negative, the function outputs a message in the example below after computing the sum of the two numbers. We must test every execution path that is conceivable in order to achieve path coverage. There are several ways to use this function.

Path 1: a > 0 (true) → result = a + b → print "Sum is positive" → return result

Path 2: a > 0 (false) → result = a – b → print "Sum is negative" → return result

We would need to create test cases that cover both paths in order to achieve path coverage. For instance:

Case 1 of the test is calculateSum(2, 3).

This test case applies to Path 1: a > 0 (true), resulting in the printing of the message "Sum is positive".

Case 2 of the test: calculateSum(-5, 2)

This test case applies to Path 2: the "Sum is negative" message will be printed if a > 0 (false).

What are the Advantages of White Box Testing?

The following list of benefits of white box testing can be summed up:

1. **Test Coverage:** White box testing enables thorough test coverage by allowing testers to inspect the internal logic, structure, and implementation details of the software.
2. **Early Issue Detection:** White box testing makes it easier to find bugs and vulnerabilities at an early stage of the development process by giving you access to the source code.
3. **Targeted Test Design:** To maximize the efficacy of testing efforts, test cases can be deliberately created to target important parts of the code.
4. **Optimizes Code and Boosts Speed:** White box testing helps find inefficient algorithms or resource-intensive operations, which optimizes code and boosts speed.
5. **Greater Understanding of the Code:** Testers develop a deeper understanding of the program architecture, allowing for more efficient troubleshooting and future maintenance.

What are the Disadvantages of White Box Testing?

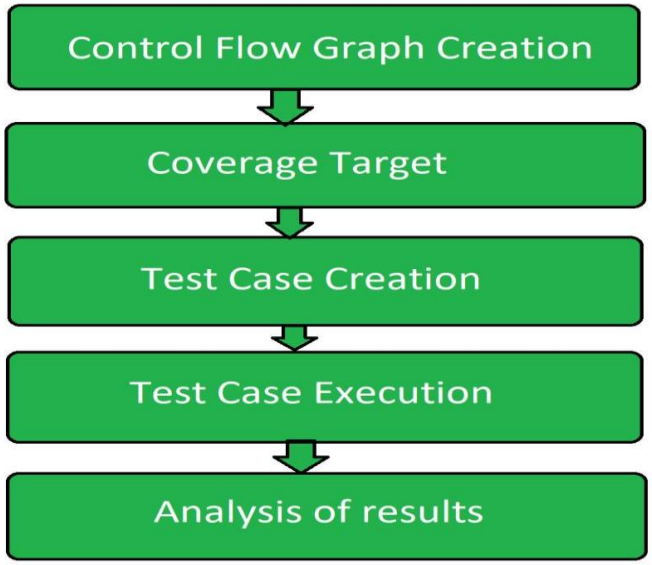
The following list of disadvantages of white box testing can be summed up:

1. **Time-Consuming**: White box testing is time-consuming because it entails examining the software's internal architecture, which takes a lot of time and effort.
2. **Test Coverage Lacking**: White box testing offers thorough coverage of the internal code, but it may ignore external aspects and user interactions, leaving the test coverage lacking.
3. **Dependence on Implementation Details:** Because white box testing is so dependent on understanding how the software functions internally, it is less successful when there are frequent code updates or when the internal organization is complicated.
4. **Ability-Required:** White box testing is only accessible to testers with significant programming abilities, making it inaccessible to non-technical testers.
5. **Insufficient Requirement Validation**: White box testing may place more emphasis on checking the functionality of the code than determining if the product satisfies the intended requirements, potentially ignoring important issues.

Control Flow Software Testing

Control flow testing is a type of software testing that uses a program's control flow as a model. Control flow testing is a structural testing strategy. This testing technique comes under white box testing. For the type of control flow testing, all the structure, design, code and implementation of the software should be known to the testing team. This type of testing method is often used by developers to test their own code and own implementation as the design, code and the implementation is better known to the developers. This testing method is implemented with the intention to test the logic of the code so that the user requirements can be fulfilled. Its main application is to relate the small programs and segments of the larger programs.

Control Flow Testing Process:
Following are the steps involved into the process of control flow testing:



Control Flow Testing Process

- **Control Flow Graph Creation:** From the given source code a control flow graph is created either manually or by using the software.
- **Coverage Target:** A coverage target is defined over the control flow graph that includes nodes, edges, paths, branches etc.
- **Test Case Creation:** Test cases are created using control flow graphs to cover the defined coverage target.
- **Test Case Execution:** After the creation of test cases over coverage target, further test cases are executed.
- **Analysis:** Analyze the result and find out whether the program is error free or has some defects.

Control Flow Graph:

Control Flow Graph is a graphical representation of control flow or computation that is done during the execution of the program. Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside a program unit. Control flow graph was originally developed by Frances E. Allen.

Cyclomatic Complexity:

Cyclomatic Complexity is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to describe the complexity of a program. It is computed using the Control Flow Graph of the program.
M = E - N + 2P

Objectives of Control Flow Testing:

- **Path Coverage:** During testing, it makes sure that every path through the program is run at least once.
- **Branch Coverage:** This guarantees that, during testing, every decision point (branch) in the program is evaluated as true or false at least once.
- **Decision Coverage:** This guarantees that all potential outcomes are covered by executing each decision point in the program at least once.
- **Loop testing:** It evaluates how the program performs in various loop scenarios, such as numerous, zero and single iterations.
- **Testing Error-Handling Paths:** It tests the program's error and exception handling pathways.
- **Multiple condition testing:** It involves testing both simple and complicated condition combinations inside decision points.
- **Boundary Value Examination:** It evaluates how the software behaves when input ranges are pushed to their limits.
- **Integration testing:** It examines how various software modules or components interact with one another.
- **Cyclomatic Complexity Reduction:** It determines and reduces the program's cyclomatic complexity.

Advantages of Control flow testing:

- It detects almost half of the defects that are determined during the unit testing.
- It also determines almost one-third of the defects of the whole program.
- It can be performed manually or automated as the control flow graph that is used can be made by hand or by using software also.

Disadvantages of Control flow testing:

- It is difficult to find missing paths if program and the model are done by same person.
- Unlikely to find spurious features.

- **Condition Coverage:** In this technique, all individual conditions must be covered as shown in the following example:
  1. READ X, Y
  2. IF(X == 0 || Y == 0)
  3. PRINT '0'
  4. #TC1 – X = 0, Y = 55
  5. #TC2 – X = 5, Y = 0

- **Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let's consider the following example:
  1. READ X, Y
  2. IF(X == 0 || Y == 0)
  3. PRINT '0'

4. #TC1: X = 0, Y = 0
5. #TC2: X = 0, Y = 5
6. #TC3: X = 55, Y = 0
7. #TC4: X = 55, Y = 5

- **Basis Path Testing:**
  In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path. **Steps:**
  1. Make the corresponding control flow graph
  2. Calculate the cyclomatic complexity
  3. Find the independent paths
  4. Design test cases corresponding to each independent path
  5. V(G) = P + 1, where P is the number of predicate nodes in the flow graph
  6. V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes
  7. V(G) = Number of non-overlapping regions in the graph
  8. #P1: 1 – 2 – 4 – 7 – 8
  9. #P2: 1 – 2 – 3 – 5 – 7 – 8
  10. #P3: 1 – 2 – 3 – 6 – 7 – 8
  11. #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

- **Loop Testing:** Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.
  1. **Simple loops:** For simple loops of size n, test cases are designed that:
     - Skip the loop entirely
     - Only one pass through the loop
     - 2 passes
     - m passes, where m < n
     - n-1 ans n+1 passes
     -
  2. **Nested loops:** For nested loops, all the loops are set to their minimum count, and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

  3. **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

White Testing is performed in 2 Steps:
1. Tester should understand the code well
2. Tester should write some code for test cases and execute them

Code Coverage :
The degree to which a certain piece of code can be evaluated, by testing the source code under a given test suite. Code coverage is thus a measurement of the extent to which a code is covered.

Statement Coverage Testing :

This is a metric which ensures that each statement of the code is executed at least once. It measures the number of lines executed. It helps to check the does and dont's of a source code. Let us understand the concept with the help of the following source code.

```
Read a;

read b;

if a>b

print "A is greater than B";

else

print "B is greater than A";

end if;
```

Condition 1 :

If a=5 and b=1, then the first print statement is executed.

Number of statements executed = 5

Total number of statements in the code = 7

Statement coverage - 5/7 * 100

Condition 2 :

If a=1 and b=5, then the second print statement is executed.

Number of statements executed = 6

Total number of statements in the code = 7

Statement coverage - 6/7 * 100

This method can be considered a white box testing, as it intends to evaluate the internal structure of the code. A programmer is the one who can perform this task efficiently.

Branch Coverage Testing:
Branch or decision coverage technique aims to test whether a program performs the requisite jump or branching. If the program branches successfully, then the rest of the code within the branch must also execute efficiently.

The diamond shapes are the decision nodes, based on the true and false conditions, the flow of program takes place.

Path Coverage Testing :
Path coverage deals with the total number of paths that could be covered by a test case. Path is actually a way, a flow of execution that follows a sequence of instructions. It covers a function from its entry till its exit point. It covers statement, branch/decision coverage. Path coverage can be understood in terms of the following :

- Loop coverage
- Function coverage
- Call coverage

Loop Coverage :
This technique is used to ensure that all the loops have been executed, and the number of times they have been executed. The purpose of this coverage technique is to make sure that the loops adhere to the conditions as prescribed and doesn't iterate infinitely or terminate abnormally. Loop testing aims at monitoring the beginning till the end of the loop.

Function Coverage :

This method assures that each function has been invoked.

Call Coverage :
It is a very common scenario in programming that one function calls another and so on. There is a calling function and a called function. So this coverage technique ensures that there does not exist any faults in function call.

Coverage Metrics :
There are various techniques for path coverage. Few of them are - Flow graphs, cyclomatic complexity, graph metrices.

- **Flow Graphs -**This is simply a graphical representation of the flow of a program.

  Each statement of a program or the decision points are represented using nodes.The

nodes that switch to another node, that is, a set of new condition(s), it is known as *independent path.*

- **Cyclomatic Complexity -**This technique follows the flow graph technique and aims to calculate the structural complexity of the code. To understand this, let's have a look at the following example. The flow of a program is simply presented with the help of a diagram. The decision nodes are known as *predicate nodes.*

  The cyclomatic complexity is calculated as follows :

  V(G) = e-n+2P

  e stands for the number of edges

  n stands for the number of nodes

  P stands for the predicate nodes

Objectives of Coverage Techniques :

- The entire concept of coverage technique focuses on bridging any gap in test cases.
- A quantitative analysis surely provides a better way to keep a track of the entire process.
- Evaluates how efficient the test cases are in delivering the right output, thereby enhancing the quality of a product.
- To what extent our program is successfully running.

Data Flow Testing

Is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams. Furthermore, it is concerned with:

- Statements where variables receive values,
- Statements where these values are used or referenced.

To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

DEF(S) = {X | statement S contains the definition of X}
USE(S) = {X | statement S contains the use of X}

If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s. Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program. Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:

- A variable is defined but not used or referenced,
- A variable is used but never defined,
- A variable is defined twice before it is used

Types of Data Flow Testing:

1. **Testing for All-Du-Paths:** It Focuses on "All Definition-Use Paths. All-Du-Paths is an acronym for "All Definition-Use Paths." Using this technique, every possible path from a variable's definition to every usage point is tested.
2. **All-Du-Path Predicate Node Testing:** This technique focuses on predicate nodes, or decision points, in the control flow graph.
3. **All-Uses Testing:** This type of testing checks every place a variable is used in the application.
4. **All-Defs Testing:** This type of testing examines every place a variable is specified within the application's code.
5. **Testing for All-P-Uses:** All-P-Uses stands for "All Possible Uses." Using this method, every potential use of a variable is tested.
6. **All-C-Uses Test:** It stands for "All Computation Uses." Testing every possible path where a variable is used in calculations or computations is the main goal of this technique.
7. **Testing for All-I-Uses:** All-I-Uses stands for "All Input Uses." With this method, every path that uses a variable obtained from outside inputs is tested.
8. **Testing for All-O-Uses:** It stands for "All Output Uses." Using this method, every path where a variable has been used to produce output must be tested.
9. **Testing of Definition-Use Pairs:** It concentrates on particular pairs of definitions and uses for variables.
10. **Testing of Use-Definition Paths:** This type of testing examines the routes that lead from a variable's point of use to its definition.

Advantages of Data Flow Testing:

Data Flow Testing is used to find the following issues-

- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.

Disadvantages of Data Flow Testing

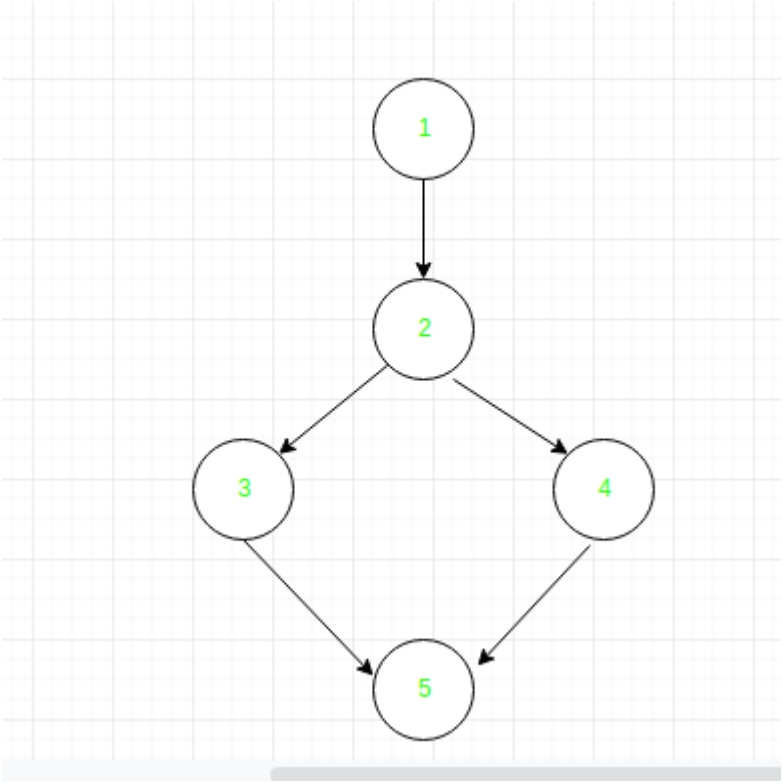
- Time consuming and costly process
- Requires knowledge of programming languages

Example:

1. read x, y;
2. if(x>y)

3. a = x+1
else
4. a = y-1
5. print a;

Control flow graph of above example:



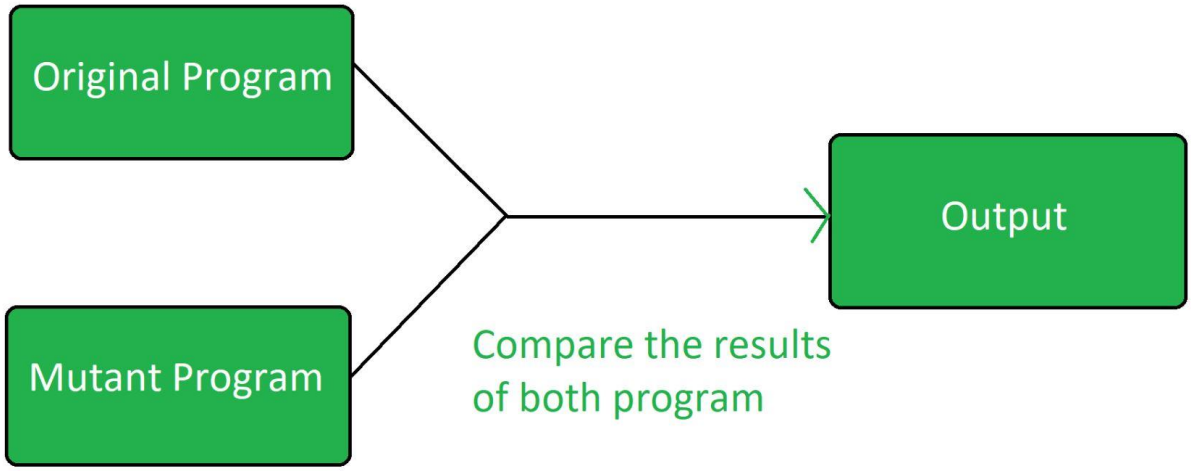
Define/use of variables of above example:

Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Mutation Testing – Software Testing

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

History of Mutation Testing:
Richard Lipton proposed the mutation testing in 1971 for the first time. Although high cost reduced the use of mutation testing but now it is widely used for languages such as Java and XML.



Mutation Testing is a White Box Testing.
Mutation testing can be applied to design models, specifications, databases, tests, and XML. It is a structural testing technique, which uses the structure of the code to guide the testing process. It can be described as the process of rewriting the source code in small ways in order to remove the redundancies in the source code.

Objective of Mutation Testing:

The objective of mutation testing is:

- To identify pieces of code that are not tested properly.
- To identify hidden defects that can't be detected using other testing methods.
- To discover new kinds of errors or bugs.
- To calculate the mutation score.
- To study error propagation and state infection in the program.
- To assess the quality of the test cases.

Types of Mutation Testing:

Mutation testing is basically of 3 types:
1. Value Mutations:
In this type of testing the values are changed to detect errors in the program. Basically a small value is changed to a larger value or a larger value is changed to a smaller value. In this testing basically constants are changed.
Example:
Initial Code:

```
int mod = 1000000007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

Changed Code:

```
int mod = 1007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

2. Decision Mutations:

In decisions mutations are logical or arithmetic operators are changed to detect errors in the program.

Example:

Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a > b)
 c = 10;
else
 c = 20;
```

3. Statement Mutations:

In statement mutations a statement is deleted or it is replaces by some other statement.

Example:

Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a < b)
 d = 10;
else
 d = 20;
```

Tools used for Mutation Testing :

- Judy
- Jester
- Jumble
- PIT
- MuClipse.

Advantages of Mutation Testing:

- It brings a good level of error detection in the program.
- It discovers ambiguities in the source code.
- It finds and solves the issues of loopholes in the program.
- It helps the testers to write or automate the better test cases.
- It provides more efficient programming source code.

Disadvantages of Mutation Testing:

- It is highly costly and time-consuming.
- It is not able for Black Box Testing.
- Some, mutations are complex and hence it is difficult to implement or run against various test cases.
- Here, the team members who are performing the tests should have good programming knowledge.
- Selection of correct automation tool is important to test the programs.

Code Coverage Testing in Software Testing

Code Coverage :
Code coverage is a software testing metric or also termed as a Code Coverage Testing which helps in determining how much code of the source is tested which helps in accessing quality of test suite and analyzing how comprehensively a software is verified. Actually in simple code coverage refers to the degree of which the source code of the software code has been tested. This Code Coverage is considered as one of the form of white box testing.
As we know at last of the development each client wants a quality software product as well as the developer team is also responsible for delivering a quality software product to the customer/client. Where this quality refers to the product's performance, functionalities, behavior, correctness, reliability, effectiveness, security, and maintainability. Where Code Coverage metric helps in determining the performance and quality aspects of any software.

The formula to calculate code coverage is

*Code Coverage = (Number of lines of code executed)/(Total Number of lines of code in a system component) * 100*

Code Coverage Criteria :
To perform code coverage analysis various criteria are taken into consideration. These are the major methods/criteria which are considered.
1. Statement Coverage/Block coverage :
The number of statements that have been successfully executed in the program source code.
*Statement Coverage = (Number of statements executed)/(Total Number of statements)*100.*

2. Decision Coverage/Branch Coverage :
The number of decision control structures that have been successfully executed in the program source code.
*Decision Coverage = (Number of decision/branch outcomes exercised)/(Total number of decision outcomes in the source code)*100.*

3. Function coverage :
The number of functions that are called and executed at least once in the source code.
*Function Coverage = (Number of functions called)/(Total number of function)*100.*

4. Condition Coverage/Expression Coverage :
The number of Boolean condition/expression statements executed in the conditional statement.
*Condition Coverage =(Number of executed operands)/(Total Number of Operands)*100.*

Advantages of Using Code Coverage :

- It helps in determining the performance and quality aspects of any software.
- It helps in evaluating quantitative measure of code coverage.
- It helps in easy maintenance of code base.
- It helps in accessing quality of test suite and analyzing how comprehensively a software is verified.
- It helps in exposure of bad, dead, and unused code.
- It helps in creating extra test cases to increase coverage.
- It helps in developing the software product faster by increasing its productivity and efficiency.
- It helps in measuring the efficiency of test implementation.
- It helps in finding new test cases which are uncovered.

Disadvantages of Using Code Coverage :

- Some times it fails to cover code completely and correctly.
- It can not guarantee that all possible values of a feature is tested with the help of code coverage.
- It fails in ensuring how perfectly the code has been covered.

Basis Path Testing in Software Testing

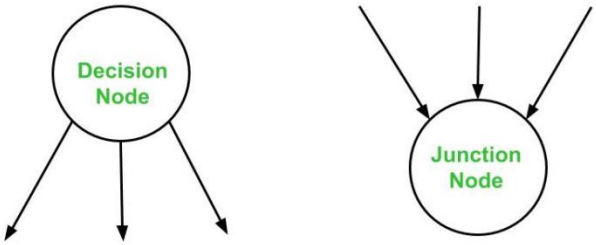
Prerequisite – Path Testing **Basis Path Testing** is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition, Basis path testing is a technique of selecting the paths in the control flow graph, that provide a basis set of execution paths through the program or module. Since this testing is based on the control structure of the program, it requires complete knowledge of the program's structure. To design test cases using this technique, four steps are followed :

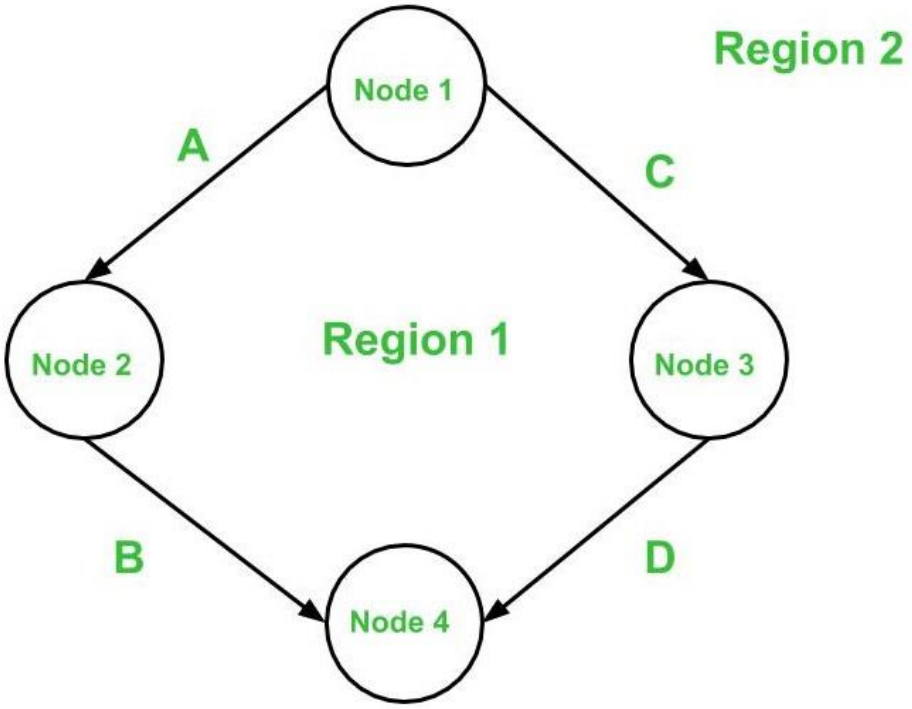
1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths

Let's understand each step one by one

. **1. Control Flow Graph –** A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

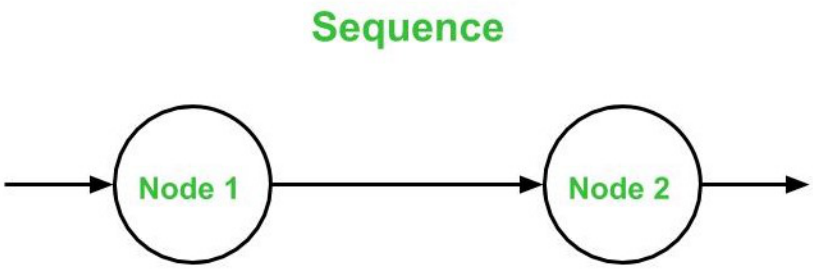
- **Junction Node –** a node with more than one arrow entering it.
- **Decision Node –** a node with more than one arrow leaving it.
- **Region –** area bounded by edges and nodes (area outside the graph is also counted as a region.).

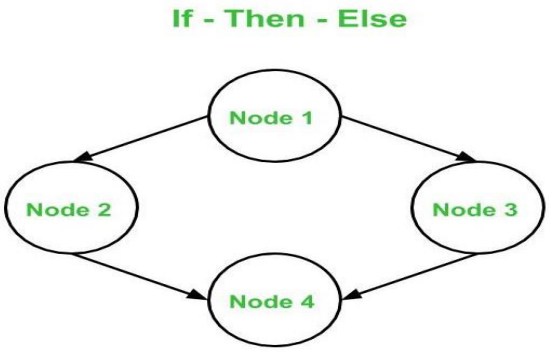




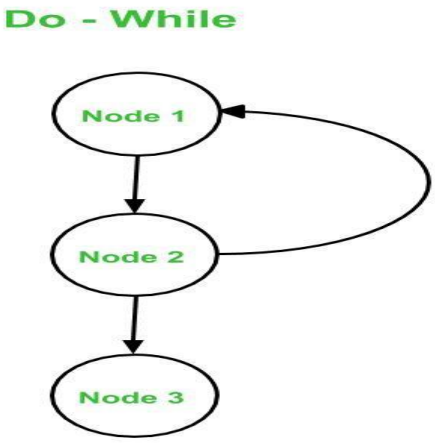
Below are the **notations** used while constructing a flow graph :

- **Sequential Statements –**

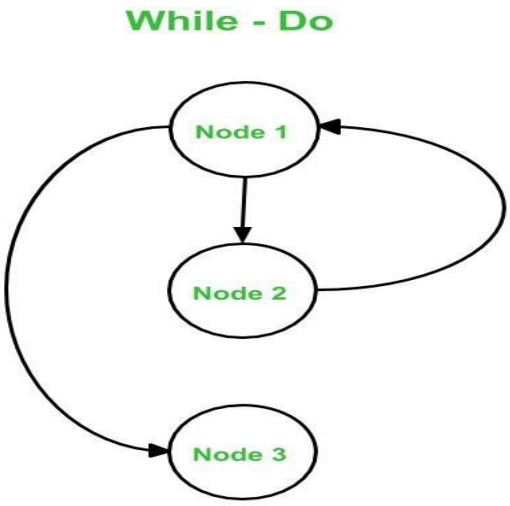


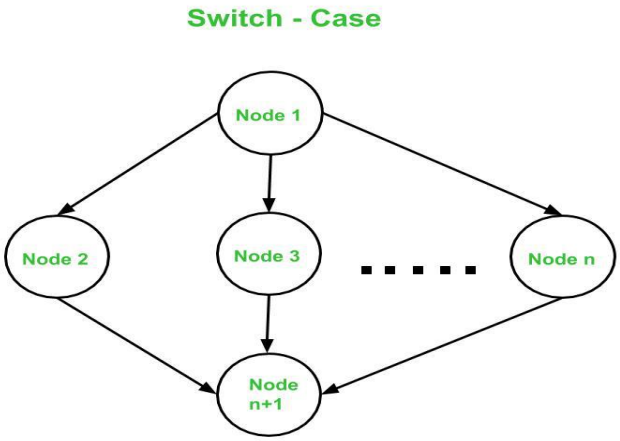


- **If – Then – Else –**
- **Do – While –**



- **While – Do –**





- **Switch – Case –**

Cyclomatic Complexity – The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

1. **Formula based on edges and nodes :**

V(G) = e - n + 2*P

1. Where, e is number of edges, n is number of vertices, P is number of connected components. For example, consider first graph given above,

where, e = 4, n = 4 and p = 1

So,

Cyclomatic complexity V(G)

= 4 - 4 + 2 * 1

= 2

1. **Formula based on Decision Nodes :**

V(G) = d + P

1. where, d is number of decision nodes, P is number of connected nodes. For example, consider first graph given above,

where, d = 1 and p = 1

So,

Cyclomatic Complexity V(G)

= 1 + 1

= 2

1. **Formula based on Regions :**

V(G) = number of regions in the graph

1. For example, consider first graph given above,

Cyclomatic complexity V(G)

= 1 (for Region 1) + 1 (for Region 2)

= 2

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph. **Note –**

1. For one function [e.g. Main( ) or Factorial( )], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of them. Also, in the cyclomatic complexity formula, the value of 'p' is set depending of the number of graphs present in total.
2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula :

d = k - 1

1. Here, k is number of arrows leaving the decision node.

Independent Paths : An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once. Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity. So, the independent paths in above first given graph :

- **Path 1:**

A -> B

- **Path 2:**

C -> D

Note – Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature. **Design Test Cases :** Finally, after obtaining the

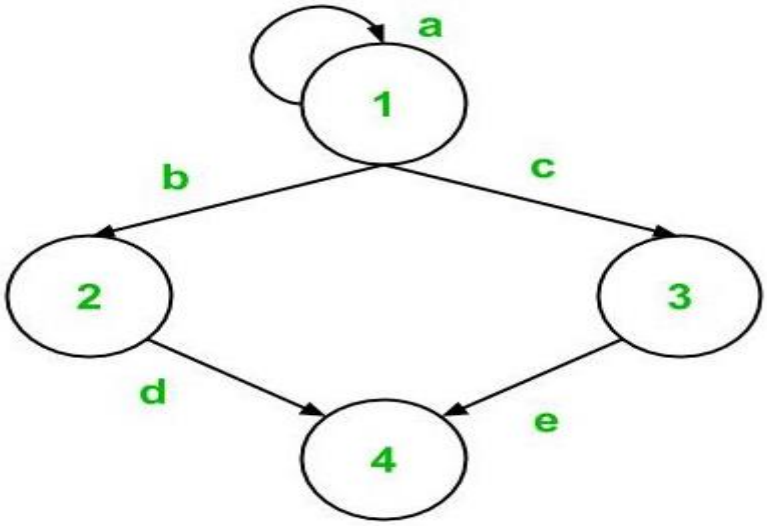
independent paths, test cases can be designed where each test case represents one or more independent paths.
Advantages : Basis Path Testing can be applicable in the following cases:

1. **More Coverage –** Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.
2. **Maintenance Testing –** When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.
3. **Unit Testing –** When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.
4. **Integration Testing –** When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.
5. **Testing Effort –** Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

Graph Matrices in Software Testing
A **graph matrix** is a data structure that can assist in developing a tool for automation of path testing. Properties of graph matrices are fundamental for developing a test tool and hence graph matrices are very useful in understanding software testing concepts and theory.

What is a Graph Matrix ?

A graph matrix is a square matrix whose size represents the number of nodes in the control flow graph. If you do not know what control flow graphs are, then read this article. Each row and column in the matrix identifies a node and the entries in the matrix represent the edges or links between these nodes. Conventionally, nodes are denoted by digits and edges are denoted by letters.
Let’s take an example.



Let's convert this control flow graph into a graph matrix. Since the graph has 4 nodes, so the graph matrix would have a dimension of 4 X 4. Matrix entries will be filled as follows :

- (1, 1) will be filled with 'a' as an edge exists from node 1 to node 1
- (1, 2) will be filled with 'b' as an edge exists from node 1 to node 2. It is important to note that (2, 1) will not be filled as the edge is unidirectional and not bidirectional
- (1, 3) will be filled with 'c' as edge c exists from node 1 to node 3
- (2, 4) will be filled with 'd' as edge exists from node 2 to node 4
- (3, 4) will be filled with 'e' as an edge exists from node 3 to node 4

The graph matrix formed is shown below :

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

Connection Matrix :

A connection matrix is a matrix defined with edges weight. In simple form, when a connection exists between two nodes of control flow graph, then the edge weight is 1, otherwise, it is 0. However, 0 is not usually entered in the matrix cells to reduce the complexity.

For example, if we represent the above control flow graph as a connection matrix, then the result would be :

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

As we can see, the weight of the edges are simply replaced by 1 and the cells which were empty before are left as it is, i.e., representing 0.

A connection matrix is used to find the **cyclomatic complexity of the control graph**. Although there are three other methods to find the cyclomatic complexity but this method works well too.
Following are the **steps to compute the cyclomatic complexity** :

1. Count the number of 1s in each row and write it in the end of the row
2. Subtract 1 from this count for each row (Ignore the row if its count is 0)
3. Add the count of each row calculated previously
4. Add 1 to this total count
5. The final sum in Step 4 is the cyclomatic complexity of the control flow graph

Let's apply these steps to the graph above to compute the cyclomatic complexity.

	1	2	3	4	
1	1	1	1		3 - 1 = 2
2				1	1 - 1 = 0
3				1	1 - 1 = 0
4					Ignore
Cyclomatic complexity = 2 + 0 + 0 + 1 = 3					

We can verify this value for cyclomatic complexity using other methods :

Method-1 :
Cyclomatic complexity

= e - n + 2 * P

Since here,

e = 5

n = 4

and, P = 1

Therefore, cyclomatic complexity,

= 5 - 4 + 2 * 1

= 3

Method-2 :
Cyclomatic complexity

= d + P

Here,

d = 2

and, P = 1

Therefore, cyclomatic complexity,

= 2 + 1

= 3

Method-3:
Cyclomatic complexity

= number of regions in the graph

li

- Region 1: bounded by edges b, c, d, and e
- Region 2: bounded by edge a (in loop)
- Region 3: outside the graph

Therefore, cyclomatic complexity,

= 1 + 1 + 1

= 3

Difference Between Black Box and White Box Testing

- n Black Box, testing is done without the knowledge of the internal structure of program or application whereas in White Box, testing is done with knowledge of the internal structure of program.
- When we compare Blackbox and Whitebox testing, Black Box test doesn’t require programming knowledge whereas the White Box test requires programming knowledge.

- Black Box testing has the main goal to test the behavior of the software whereas White Box testing has the main goal to test the internal operation of the system.
- Comparing White box testing and Black box testing, Black Box testing is focused on external or end-user perspective whereas White Box testing is focused on code structure, conditions, paths and branches.
- Black Box test provides low granularity reports whereas the White Box test provides high granularity reports.
- Comparing Black box testing vs White box testing, Black Box testing is a not time-consuming process whereas White Box testing is a time-consuming process

What is Black Box testing?

In Black-box testing, a tester doesn't have any information about the internal working of the software system. Black box testing is a high level of testing that focuses on the behavior of the software. It involves testing from an external or end-user perspective. Black box testing can be applied to virtually every level of software testing: unit, integration, system, and acceptance.

What is White Box testing?

White-box testing is a testing technique which checks the internal functioning of the system. In this method, testing is based on coverage of code statements, branches, paths or conditions. White-Box testing is considered as low-level testing. It is also called glass box, transparent box, clear box or code base testing. The white-box Testing method assumes that the path of the logic in a unit or program is known.

Parameter	Black Box testing	White Box testing
Definition	It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.

Parameter	Black Box testing	White Box testing
Alias	It also knowns as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
Implementation knowledge	Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
Objective	The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.

Parameter	Black Box testing	White Box testing
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.
Code Access	Code access is not required for Black Box Testing.	White box testing requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can bring in hidden defects.
Skill level	Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform white box testing.

Parameter	Black Box testing	White Box testing
Techniques	Equivalence partitioning is Black box testing technique is used for Blackbox testing. Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage, Branch coverage, and Path coverage are White Box testing technique. Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing.

What is Black Box Testing in SDLC?

Software testing methodology known as “Black Box Testing” is employed in the discipline of software engineering. Without requiring knowledge of a software system’s core code or implementation details, it focuses on evaluating its functionality. In other words, the tester examines the system’s inputs and outputs as a “black box,” without having any knowledge of how the system processes the inputs or generates the outputs.

Black box testing’s objective is to find flaws or mistakes in the system’s behavior by giving it different inputs and watching what happens. With this technique, the system is largely validated against its functional specifications, user expectations, and stated needs. It makes sure that the program runs properly from the user’s point of view.

What is the scope of Black Box Testing?

Black box testing makes sure to monitor the input that the software receives while examining the desired outcome. The range of Black box testing is shown below.

- Test your software quickly to ensure the intended purpose is realized.
- Without any prior coding experience with the application, testers can complete this task.
- It aids in testing the software's functionality.

What are the Characteristics of Black Box Testing?

Black box testing's salient features include:

1. **Lack of Internal Structure Knowledge:** When conducting black box testing, testers are not given access to the software system's source code, internal architecture, or implementation specifics. They only rely on the behavior and external interfaces of the system.
2. Black box testing places a strong emphasis on validating the software system against its requirements and making sure that it accurately carries out the expected functions.
3. **Testing Based on Expected Inputs and Outputs:** When creating test cases, testers take into account the expected inputs and outputs that the system should generate. When designing test cases, the internal operations of the system are not taken into account.
4. **Based on Equivalence Partitions and Border Values, the Following Test Cases:** Black box testing frequently uses methods like boundary value analysis and equivalence partitioning to create efficient test cases. Boundary value analysis focuses on evaluating values near the edges of these partitions whereas equivalence partitioning divides the input space into groups with comparable behavior.
5. **Verification of Functional and Non-Functional Aspects:** Black box testing includes both functional and non-functional components of the software system, such as performance, reliability, and usability. Examples of functional aspects include user interface, data validation, and error handling.
6. **Independently Performed:** Testers who are not members of the development team can carry out black-box testing. With this independence, the behavior of the software may be evaluated objectively.

What kinds of Black Box Testing are There?

Black box testing can be applied to functional, non-functional, and regression testing, which are the three primary types of tests.

Functional Testing:

Specific features or operations of the software that is being tested can be tested via black box testing. Make sure, for instance, that only the proper user credentials may be used to log in and that the wrong ones cannot. Functional testing might concentrate on the most important features of the software (smoke testing/sanity testing), on how well the system works as a whole (system testing), or on the integration of its essential components.

Non-Functional Testing:

Black box testing enables the examination of additional software components in addition to features and functionality. Non-functional tests focus more on "how" than "if" the software can complete a task. Black box tests reveal whether the software is:

- Accessible to and understandable by its consumers
- Effective under-anticipated or peak loads
- Appropriate for the relevant screen sizes, browsers, operating systems, and/or devices
- Exposed to common security threats or security vulnerabilities.

Regression Testing:

Black box testing can be used to detect whether a new software version exhibits a regression, or a reduction in capabilities, from one version to the next. Regression testing can be used to test both

functional and non-functional features of the software, such as when a particular feature no longer functions as expected in the new version or when a formerly fast-performing action becomes much slower in the new version.

What are the different types of techniques used in Black Box Testing?

Black box testing is a method of software testing where the tester does not know the internal structure, particulars of the design, or details of the implementation of the system being tested. Instead, the tester ignores the system's internal workings and instead concentrates on its inputs, outputs, and behavior. Here are a few typical methods for black box testing:

Here are a few typical methods for black box testing:

- **Equivalence Partitioning:** This technique involves grouping or partitioning the input data on the grounds that if one test case in a partition shows a flaw, subsequent test cases in the same partition are probably going to behave in the same way.
- **Boundary Value Analysis**: In this method, test cases are created to assess how the system behaves when input domain boundaries are reached. Testing professionals can improve their chances of discovering flaws by concentrating on values close to the borders, where errors frequently occur.
- **Testing Using Decision Tables:** Decision tables are used to represent intricate logical connections between inputs and outputs. The combinations of input conditions and related system actions are used to create test cases. By using this method, it is possible to make sure that every combination is covered.
- **State Transition Testing:** When a system's behavior is dependent on its present state and inputs, state transition testing is utilized. Test cases are created to simulate various state changes and verify the system's response at each one.
- **Error Guessing:** Testers make educated guesses about possible flaws or mistakes in the system based on their knowledge and intuition. Based on this understanding, test cases are created to find flaws that would not be detected by other methods.
- **All Pair Testing:** Testing of all pairs Technique is employed to test all conceivable discrete value combinations. This combinational approach is used to test applications that incorporate input from checkboxes, radio buttons, list boxes, text boxes, etc.
- **Cause-Effect:** Cause-Effect The relationship between a particular result and all the factors influencing the result is highlighted by technique. It is founded on a number of conditions.
- **Use Case:** Case study Technique for locating test cases throughout the system, from the start to the finish, according to usage. Using this method, the testing team develops a test scenario that can put the entire piece of software through its paces based on how well each function works from beginning to end.

Boundary Value Analysis

Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.

It checks for the input values near the boundary that have a higher chance of error. Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition

- A boundary value for a valid partition is a valid boundary value.
- A boundary value for an invalid partition is an invalid boundary value.
- For each variable we check-
    - Minimum value.
    - Just above the minimum.
    - Nominal Value.
    - Just below Max value.
    - Max value.

Example: Consider a system that accepts ages from 18 to 56
.

Boundary Value Analysis(Age accepts 18 to 56)		
Invalid (min-1)	Valid (min, min + 1, nominal, max – 1, max)	Invalid (max + 1)
17	18, 19, 37, 55, 56	57

Valid Test cases: Valid test cases for the above can be any value entered greater than 17 and less than 57.

- Enter the value- 18.
- Enter the value- 19.
- Enter the value- 37.
- Enter the value- 55.
- Enter the value- 56.

Invalid Testcases: When any value less than 18 and greater than 56 is entered.

- Enter the value- 17.
- Enter the value- 57.

Single Fault Assumption: When more than one variable for the same application is checked then one can use a single fault assumption. Holding all but one variable to the extreme value and allowing the remaining variable to take the extreme value. For n variable to be checked:

Maximum of 4n+1 test cases

Problem: Consider a Program for determining the Previous Data.

Input: *Day, Month, Year with valid ranges as-*

$1 \leq Month \leq 12$

$1 \leq Day \leq 31$

$1900 \leq Year \leq 2000$

Design Boundary Value Test Cases.

Solution: Taking the year as a Single Fault Assumption i.e. year will be having values varying from 1900 to 2000 and others will have nominal values.

Test Cases	Month	Day	Year	Output
1	6	15	1990	14 June 1990
2	6	15	1901	14 June 1901
3	6	15	1960	14 June 1960
4	6	15	1999	14 June 1999
5	6	15	2000	14 June 2000

Taking Day as Single Fault Assumption i.e. Day will be having values varying from 1 to 31 and others will have nominal values.

Test Case	Month	Day	Year	Output
6	6	1	1960	31 May 1960
7	6	2	1960	1 June 1960
8	6	30	1960	29 June 1960
9	6	31	1960	Invalid day

Taking Month as Single Fault Assumption i.e. Month will be having values varying from 1 to 12 and others will have nominal values.

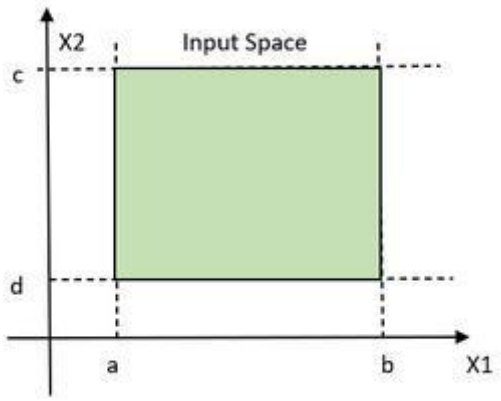
Test Case	Month	Day	Year	Output
10	1	15	1960	14 Jan 1960
11	2	15	1960	14 Feb 1960
12	11	15	1960	14 Nov 1960

Test Case	Month	Day	Year	Output
13	12	15	1960	14 Dec 1960

For the n variable to be checked Maximum of 4n + 1 test case will be required. Therefore, for n = 3, the maximum test cases are-

4 × 3 + 1 =13

The focus of BVA: BVA focuses on the input variable of the function. Let's define two variables X1 and X2, where X1 lies between a and b and X2 lies between c and d.



Showing legitimate domain

The idea and motivation behind BVA are that errors tend to occur near the extremes of the variables. The defect on the boundary value can be the result of countless possibilities.

Typing of Languages: BVA is not suitable for free-form languages such as COBOL and FORTRAN, These languages are known as weakly typed languages. This can be useful and can cause bugs also.
PASCAL, ADA is the strongly typed language that requires all constants or variables defined with an associated data type.

Limitation of Boundary Value Analysis:

- It works well when the product is under test.
- It cannot consider the nature of the functional dependencies of variables.
- BVA is quite rudimentary.

Equivalence Partitioning

It is a type of black-box testing that can be applied to all levels of software testing. In this technique, input data are divided into the equivalent partitions that can be used to derive test cases-

- In this input data are divided into different equivalence data classes.
- It is applied when there is a range of input values.

Example: Below is the example to combine Equivalence Partitioning and Boundary Value.

Consider a field that accepts a minimum of 6 characters and a maximum of 10 characters. Then the partition of the test cases ranges 0 – 5, 6 – 10, 11 – 14.

Test Scenario	Test Description	Expected Outcome
1	Enter value 0 to 5 character	Not accepted
2	Enter 6 to 10 character	Accepted
3	Enter 11 to 14 character	Not Accepted

Why Combine Equivalence Partitioning and Boundary Analysis Testing: Following are some of the reasons why to combine the two approaches:

- In this test cases are reduced into manageable chunks.
- The effectiveness of the testing is not compromised on test cases.
- Works well with a large number of variables.

Robustness Testing

Robustness testing is a quality assurance methodology focussed on testing the robustness of the software and helps in removing the reliability of the software by finding the corner cases by inputting the data that mimics the extreme environmental conditions to determine whether or not the system is robust enough to deliver the required functionality to the user.

What is Robustness in Software testing?

Robustness is a measure of how well a software system can cope with invalid inputs or unexpected user interactions. A robust system is one that continues to function correctly even in the face of unexpected or invalid inputs. A software system that is robust is able to handle errors and unexpected inputs gracefully, without crashing or producing incorrect results. It is also able to adapt to changes in its operating environment, such as changes in the operating system, hardware, or other software components.

- Robustness is an important quality for any software system, but it is especially important for systems that are safety-critical or mission-critical.
- For these systems, a failure could have serious consequences, so it is essential that they be able to handle any unexpected inputs or conditions gracefully.

What is Robustness Testing?

Robustness testing is a type of testing that is performed to assess the ability of a system or component to function correctly when it is subjected to invalid or unexpected inputs, or when it is operating outside of its specified operating conditions. It is typically used to test for the presence of memory leaks or other types of errors that can cause a system to crash.

Robustness testing is also sometimes referred to as reliability testing, stress testing, or endurance testing.

- The purpose of robustness testing is to identify the parts of the system that are most vulnerable to failure and to determine how the system can be made more resistant to failure.
- Robustness testing is typically conducted by subjecting the system to a variety of stressful conditions, such as high temperatures, high humidity, high pressure, and high levels of vibration.
- Robustness testing is typically conducted during the later stages of software testing after the software has been shown to work correctly under normal conditions.
- A common example of robustness testing is testing how a system responds to unexpected input values.
- For example, if a system is designed to accept numerical input values between 1 and 10, a robustness test would involve trying to input values outside of this range, such as 0, 11, or -5, to see how the system responds.
- Another example of robustness testing would be testing how a system responds to unexpected environmental conditions, such as excessive heat, cold, or humidity.

Why is Robustness Testing important?

- **Handle Unexpected Inputs:** Robustness testing is important because it helps ensure that a system can handle unexpected or abnormal inputs without crashing.
- **Uncover Potential Issues:** This type of testing can help uncover potential issues that could cause a system to fail in unexpected ways. By uncovering these issues early on, they can be fixed before the system is put into production.
- **Test Limits:** It allows developers to test the limits of their software and ensure that it can handle unexpected inputs and situations.
- **Uncover Hidden Bugs:** This type of testing can help uncover hidden bugs that could cause major problems in the field.
- **Stability:** This can help to prevent software failures and crashes, and can also help to improve the overall stability of your software.

Robustness Testing Scenarios

- **Navigation:** In this scenario, the tester will try to break the navigation of the application by trying to access pages or features that are not intended to be accessed by the user. This can be done by trying to access pages directly through the URL or by trying to access hidden features by manipulating the application's code.
- **Functionality:** In this scenario, the tester will try to break the functionality of the application by trying to input invalid data or by trying to perform actions that are not intended to be performed by the user. This can be done by trying to submit invalid data through the application's forms or by trying to access hidden features by manipulating the application's code.
- **Security:** In this scenario, the tester will try to break the security of the application by trying to exploit known vulnerabilities or by trying to gain access

to sensitive data. This can be done by trying to inject SQL code into the application's forms or by trying to access restricted areas of the application.
- **Invalid input:** The system should be able to handle invalid input gracefully. For example, if a user enters an invalid command, the system should display an error message and prompt the user for another input.
- **Interfaces with new software modules:** The system should be able to interface with new software modules without any issues. For example, if a new software module is added to the system, the system should be able to communicate with it without any errors.
- **Changes in the environment:** The system should be able to function properly in different environments. For example, if the system is moved from one location to another, it should be able to function properly in the new location.
- **External environment changes affecting the hardware system:** The system should be able to function properly even if the external environment changes affect the hardware system. For example, if the temperature or humidity changes, the system should be able to function properly.

Types of Testing to ensure Robustness of Test Suites

There are many types of testing that can ensure the robustness of test suites. Below are some of the testing types that ensure the robustness of test suites:

- **Regression testing:** Regression testing is a type of testing that is used to find bugs in software that have already been fixed. This is done by running the software with different inputs and comparing the output to the expected output. This type of testing is used to verify that a software program continues to function properly after it has been modified or updated. This type of testing is typically performed after a new version of the software has been released, or after a change has been made to the code.
- **Functional testing:** Functional testing is a type of testing that is used to verify that a system or software performs as expected. Load testing is a type of testing that is used to verify that a system or software can handle a heavy load or traffic. This type of testing is typically performed by running the software through a series of tests that exercise the various functions of the software.
- **Load testing:** Load testing is a type of testing that is used to find bugs in software by running it with different inputs and checking if the output is as expected. It is used to verify that a software program is able to handle the load that is expected to be placed on it. This type of testing is typically done by running the software through a series of tests that simulate the load that the software will experience in production.
- **Stress testing:** Stress testing involves subjecting a system to intense or extreme conditions in order to see how well it holds up. This can help to identify potential issues that may only arise under high levels of stress or strain.
- **Negative testing:** Negative testing involves deliberately providing invalid or incorrect inputs to a system in order to see how it responds. This can help to uncover errors in input validation or handling that could lead to security vulnerabilities or data loss.
- **Fuzz testing:** A fuzz test (also known as a fuzzing test) is a software testing technique, usually automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The

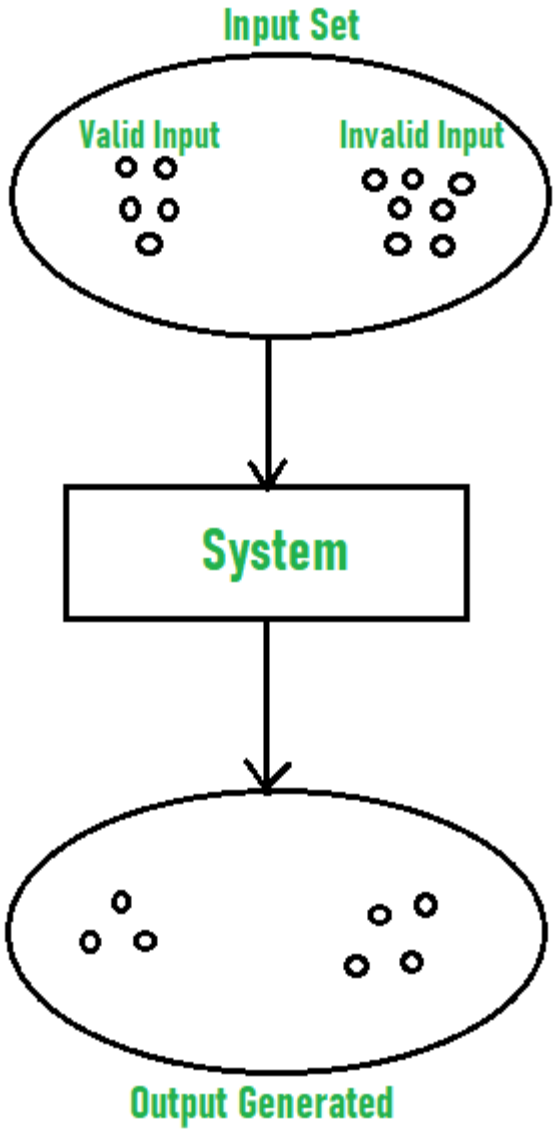
program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Fuzz testing is effective in finding coding errors and security vulnerabilities. This involves feeding random or invalid data into a system and seeing how it responds. This can help to find potential vulnerabilities in the system.

- **Use case testing**: Use case testing involves testing the system with realistic scenarios to see how it responds. This can help to identify potential usability issues.
- **Security testing:** Security Testing involves testing the system for security vulnerabilities. This can help to identify potential security risks.
- **Black-box testing:** Black-box testing is a method of software testing that examines the functionality of a software program without knowing the internal code structure. The tester is only aware of what the software is supposed to do but not how it does it. Black-box testing can be used to test the functionality of a software program, the usability of a user interface, and the compliance of a program with external standards.
- **Mutation testing:** Mutation testing is a type of software testing that involves modifying a program's source code or its inputs and then testing to see if the program still behaves as expected. The goal of mutation testing is to find faults in a program's code or inputs that can cause the program to produce incorrect results.
- **Fault injection testing:** Fault injection testing is a method of testing software by Introducing faults into the software program to see if the program can detect and handle the faults. Fault injection can be used to test the robustness of a program's error-handling capabilities.

Equivalence Partitioning Method is also known as Equivalence class partitioning (ECP). It is a software testing technique or black-box testing that divides input domain into classes of data, and with the help of these classes of data, test cases can be derived. An ideal test case identifies class of error that might require many arbitrary test cases to be executed before general error is observed.
In equivalence partitioning, equivalence classes are evaluated for given input conditions. Whenever any input is given, then type of input condition is checked, then for this input conditions, Equivalence class represents or describes set of valid or invalid states.

Guidelines for Equivalence Partitioning :

- If the range condition is given as an input, then one valid and two invalid equivalence classes are defined.
- If a specific value is given as input, then one valid and two invalid equivalence classes are defined.
- If a member of set is given as an input, then one valid and one invalid equivalence class is defined.
- If Boolean no. is given as an input condition, then one valid and one invalid equivalence class is defined.



Example-1:
Let us consider an example of any college admission process. There is a college that gives admissions to students based upon their percentage.
Consider percentage field that will accept percentage only between 50 to 90 %, more and even less than not be accepted, and application will redirect user to an error page. If percentage entered by user is less than 50 %or more than 90 %, that equivalence partitioning method will show an invalid percentage. If percentage entered is between 50 to 90 %, then equivalence partitioning method will show valid percentage.

Equivalence Partitioning		
Invalid	Valid	Invalid
<=50	50-90	>=90

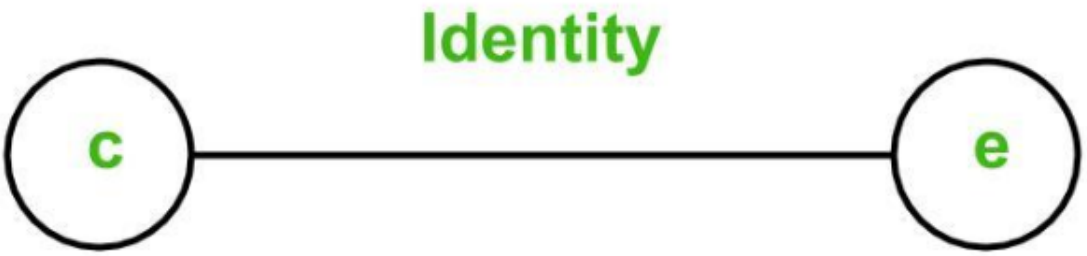
CAUSE EFFECT GRAPHING:

Cause Effect Graphing based technique is a technique in which a graph is used to represent the situations of combinations of input conditions. The graph is then converted to a decision table to obtain the test cases. Cause-effect graphing technique is used because boundary value analysis and equivalence class partitioning methods do not consider the combinations of input conditions. But since there may be some critical behaviour to be tested when some combinations of input conditions are considered, that is why cause-effect graphing technique is used. **Steps used in deriving test cases using this technique are:**

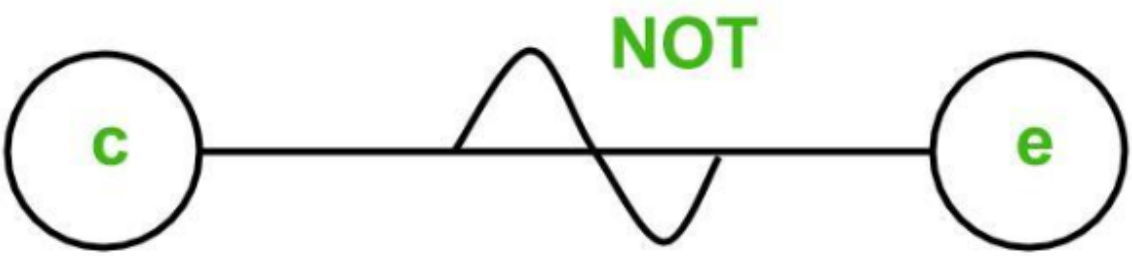
1. **Division of specification:** Since it is difficult to work with cause-effect graphs of large specifications as they are complex, the specifications are divided into small workable pieces and then converted into cause-effect graphs separately.
2. **Identification of cause and effects:** This involves identifying the causes(distinct input conditions) and effects(output conditions) in the specification.
3. **Transforming the specifications into a cause-effect graph:** The causes and effects are linked together using Boolean expressions to obtain a cause-effect graph. Constraints are also added between causes and effects if possible.
4. **Conversion into decision table:** The cause-effect graph is then converted into a limited entry decision table. If you're not aware of the concept of decision tables,
5. **Deriving test cases:** Each column of the decision-table is converted into a test case.

Basic Notations used in Cause-effect graph: Here **c** represents **cause** and **e** represents **effect**. The following notations are always **used between a cause and an effect**:

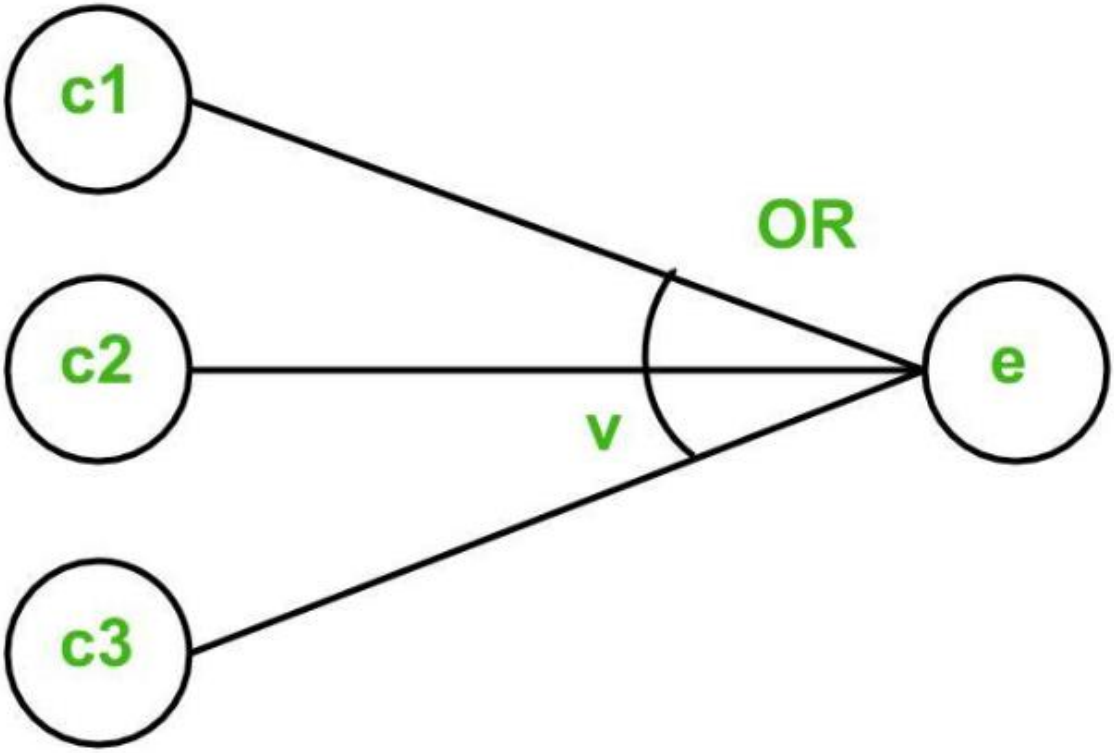
1. **Identity Function:** if c is 1, then e is 1. Else e is 0.



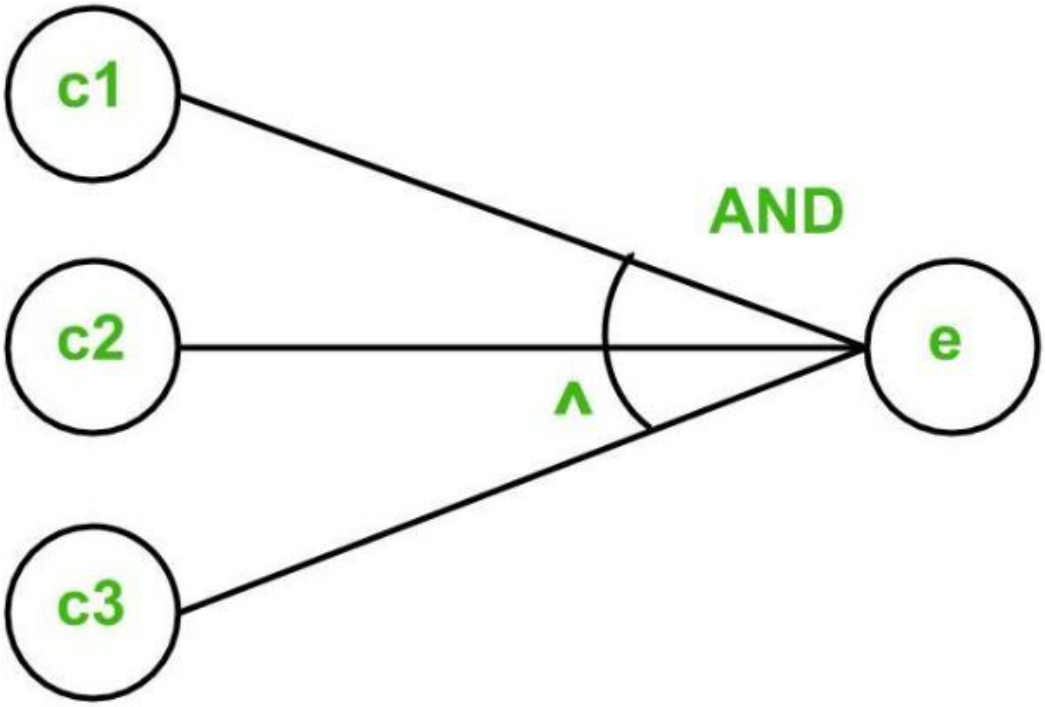
2. **NOT Function:** if c is 1, then e is 0. Else e is 1.



3. **OR Function:** if c1 or c2 or c3 is 1, then e is 1. Else e is 0.

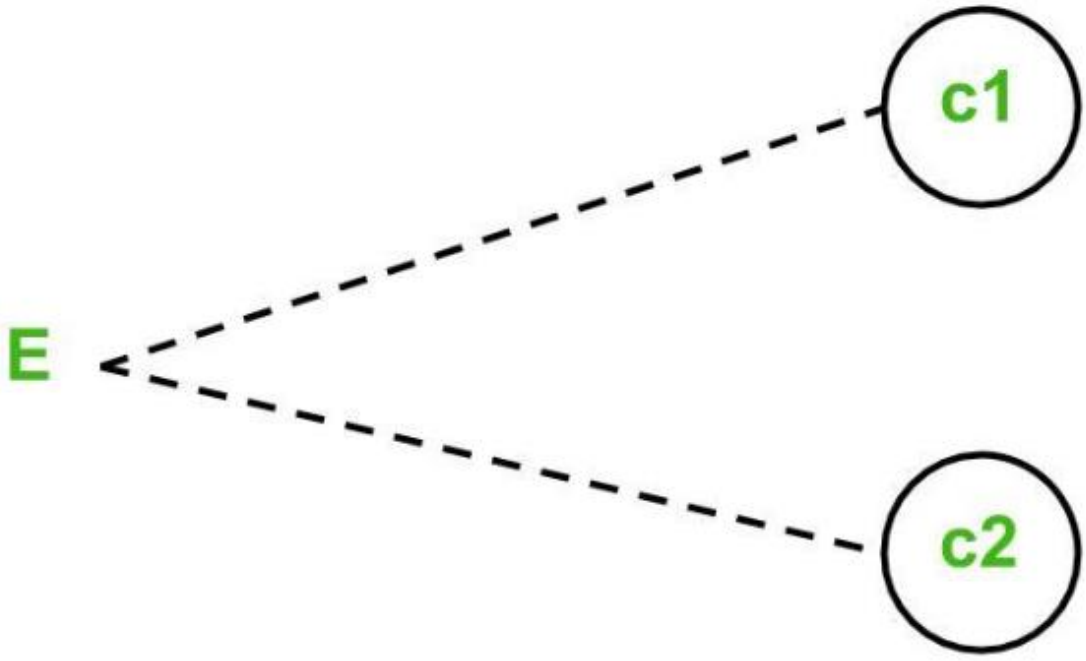


4. **AND Function:** if both c1 and c2 and c3 is 1, then e is 1. Else e is 0.

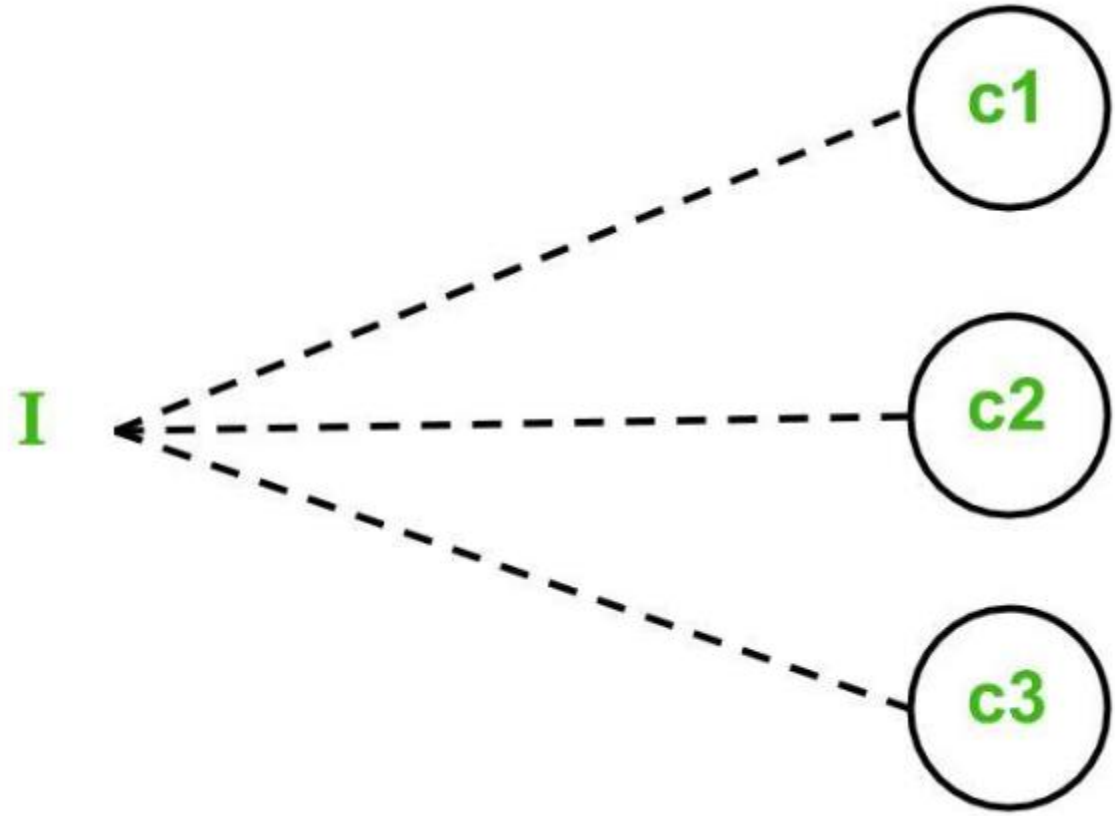


To represent some impossible combinations of causes or impossible combinations of effects, constraints are used. The following **constraints** are used in cause-effect graphs:

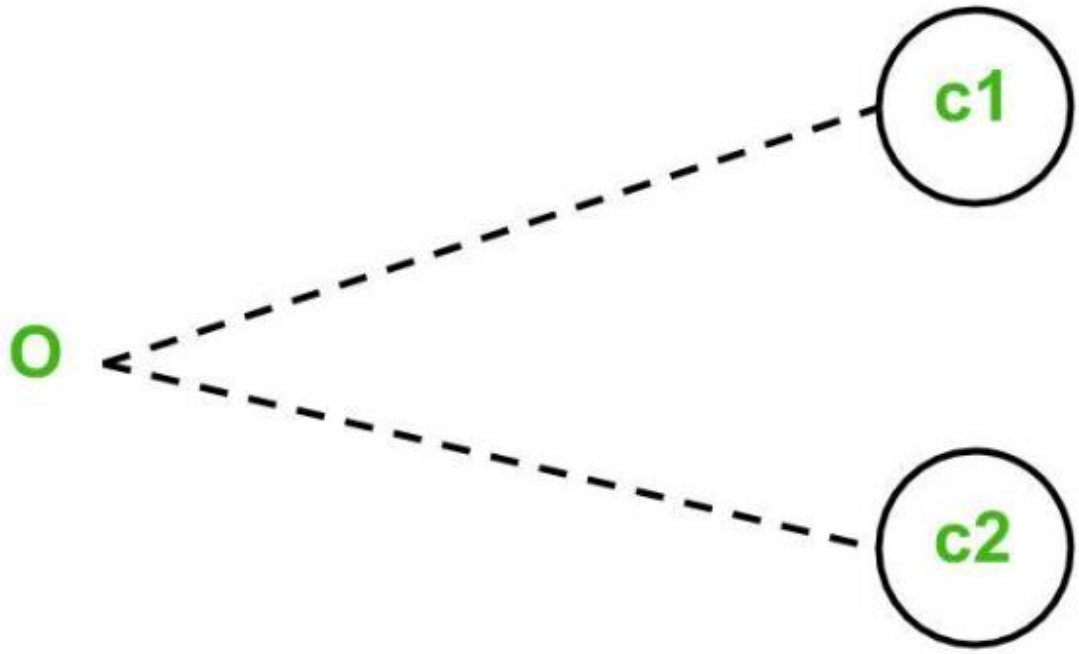
1. **Exclusive constraint** or **E-constraint:** This constraint exists between causes. It states that either c1 or c2 can be 1, i.e., c1 and c2 cannot be 1 simultaneously.



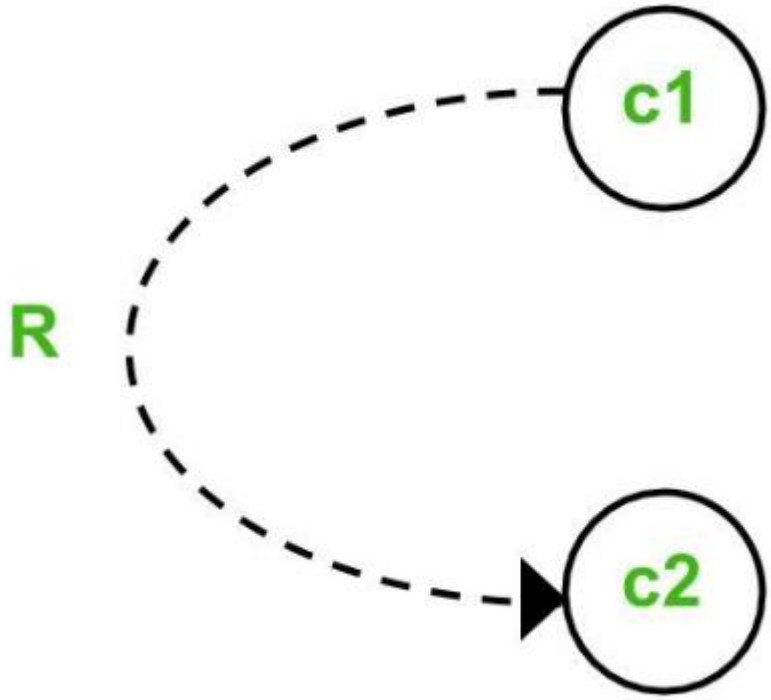
2. **Inclusive constraint** or **I-constraint:** This constraint exists between causes. It states that atleast one of c1, c2 and c3 must always be 1, i.e., c1, c2 and c3 cannot be 0 simultaneously.



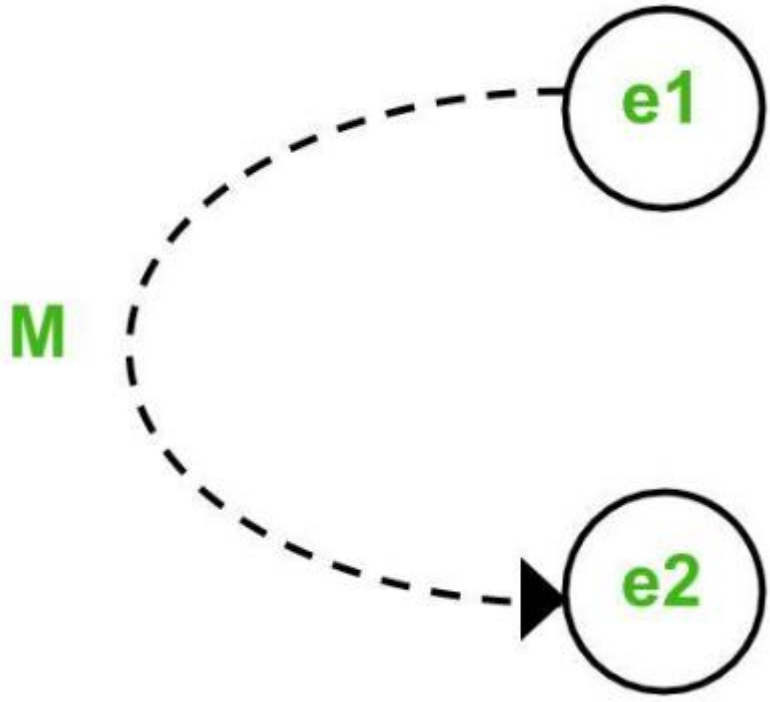
3. **One and Only One constraint** or **O-constraint:** This constraint exists between causes. It states that one and only one of c1 and c2 must be 1.



4. **Requires constraint** or **R-constraint:** This constraint exists between causes. It states that for c1 to be 1, c2 must be 1. It is impossible for c1 to be 1 and c2 to be 0.



5. **Mask constraint** or **M-constraint:** This constraint exists between effects. It states that if effect e1 is 1, the effect e2 is forced to be 0.



What is Syntax Testing?

Syntax Testing, a black box testing technique, involves testing the System inputs and it is usually automated because syntax testing produces a large number of tests.

Syntax Testing in Software Testing means it is widely used software testing term. It is done in White Box Testing by using some tools or by manually depending on the nature of the project. As you know **Syntax Testing** is used in White Box Testing so it is obviously done by developers.

Not in all the situations it can be done by developers, it can also be done by the testers if they are skilled testers mean white box testers.

As developers are doing this testing Therefore the developers should be responsible for running a syntax check before releasing their code to QA team.

In this testing, we test the syntax of the programming languages. As syntax of the every programming languages is almost different so criteria for doing Syntax Testing is also different on these programming languages.

Internal and external inputs have to conform the below formats:

- Format of the input data from users.
- File formats.
- Database schemas.

Syntax Testing - Steps:

- Identify the target language or format.
- Define the syntax of the language.
- Validate and Debug the syntax.

Syntax Testing - Limitations:

- Sometimes it is easy to forget the normal cases.
- Syntax testing needs driver program to be built that automatically sequences through a set of test cases usually stored as data.

What Is Error Guessing Technique?

Error Guessing is a Software Testing technique on guessing the error which can prevail in the code.

It is an experience-based testing technique where the Test Analyst uses his/her experience to guess the problematic areas of the application. This technique necessarily requires skilled and experienced testers.

It is a type of Black-Box Testing technique and can be viewed as an unstructured approach to Software Testing.

Error Guessing Technique

The test cases for finding issues in the software are written based on the prior testing experience with similar applications. So, the scope of the test cases would generally depend upon the kind of testing Test Analyst was involved in the past. The Error Guessing technique does not follow any specific rules.

For Example, if the Analyst guesses that the login page is error-prone, then the testers will write detailed test cases concentrating on the login page. Testers can think of a variety of combinations of data to test the login page.
To design test cases based on the Error Guessing technique, the Analyst can use past experiences to identify the conditions.

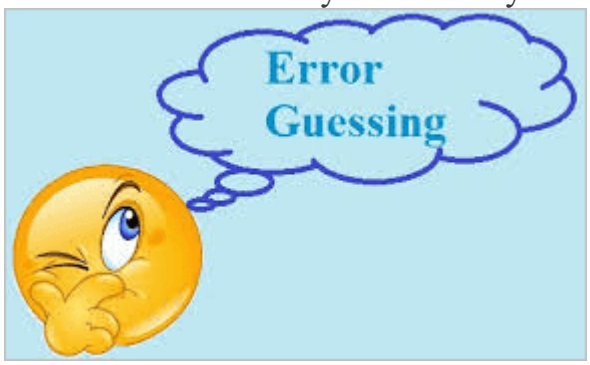
This technique can be used at any level of testing and for testing the common mistakes like:

- Divide by zero
- Entering blank spaces in the text fields
- Pressing the submit button without entering values.
- Uploading files exceeding maximum limits.
- Null pointer exception.
- Invalid parameters

The achievement rate of this technique does mainly depends upon the ability of testers.

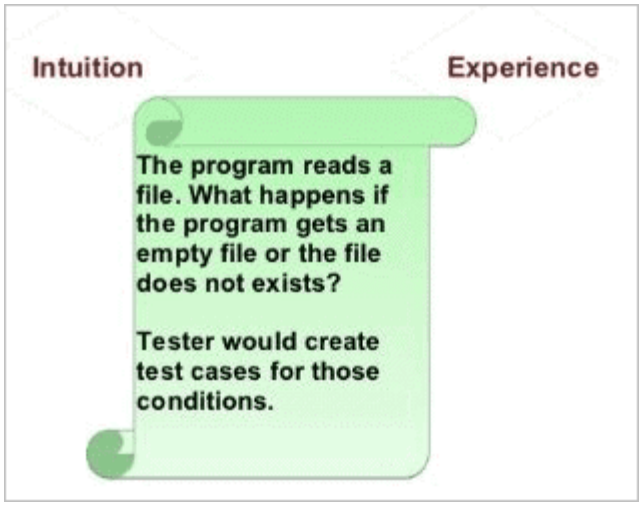
Purpose Of Error Guessing In Software Testing

- The main purpose of this technique is to guess possible bugs in the areas where formal testing would not work.
- It should obtain an all-inclusive set of testing without any skipped areas, and without creating redundant tests.
- This technique compensates for the characteristic incompleteness of Boundary Value Analysis and Equivalence Partitioning techniques.



Factors Used To Guess The Errors

Error Guessing technique requires skilled and experienced tester. It is mainly based on intuition and experience.



Following factors can be used to guess the errors:

- Lessons learned from past releases
- Tester's intuition
- Historical learning
- Previous defects
- Production tickets
- Review checklist
- Application UI
- Previous test results
- Risk reports of the application
- Variety of data used for testing.
- General testing rules
- Knowledge about AUT

When to perform Error Guessing?

It should be usually performed once most of the formal testing techniques have been applied.

Guidelines For Error Guessing

- **Remember previously troubled areas:** During any of your testing assignments, whenever you come across an interesting bug, note it down for your future reference. There are generally some common errors that happen in a specific type of application. Refer to the list of common errors for the type of application you are working on.
- **Improve your technical understanding:** Check how the code is written and how the concepts like a null pointer, arrays, indexes, boundaries, loops, etc are implemented in the code.
- Gain knowledge of the technical environment (server, operating system, database) in which the application is hosted.
- Do not just look for errors in the code but also look for errors and ambiguity in requirements, design, build, testing and usage.
- Understand the system under test
- Evaluate historical data and test results

- Keep awareness of typical implementation errors

Procedure For Error Guessing Technique

Error Guessing is fundamentally an intuitive and ad-hoc process; hence it is very difficult to give a well-defined procedure to this technique. The basic way is to first list all possible errors or error-prone areas in the application and then create test cases based on that list.

Error Guessing Example

Suppose there is a requirement stating that the mobile number should be numeric and not less than 10 characters. And, the software application has a mobile no. field.

Now, below are the Error Guessing technique:

- What will be the result if the mobile no. is left blank?
- What will be the result if any character other than a numeral is entered?
- What will be the result if less than 10 numerals are entered?

Advantages of Error Guessing technique

- Proves to be very effective when used in combination with other formal testing techniques.
- It uncovers those defects which would otherwise be not possible to find out, through formal testing. Thus, the experience of the tester saves a lot of time and effort.
- Error guessing supplements the formal test design techniques.
- Very helpful to guess problematic areas of the application.

Drawbacks of Error Guessing technique

- The focal shortcoming of this technique is that it is person dependent and thus the experience of the tester controls the quality of test cases.
- It also cannot guarantee that the software has reached the expected quality benchmark.
- Only experienced testers can perform this testing. You can’t get it done by freshers.

Conclusion

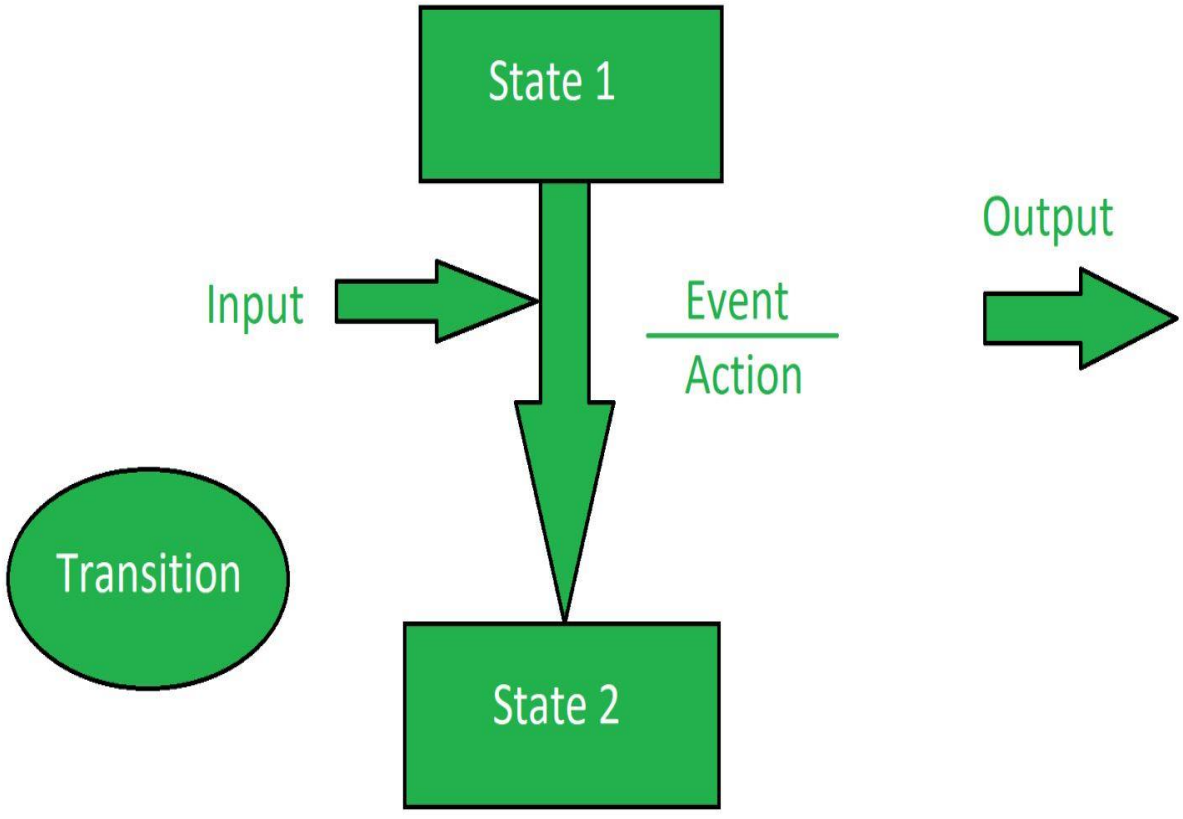
Though **Error Guessing** is one of the key techniques of testing, it does not provide full coverage of the application. It also cannot guarantee that the software has reached the expected quality benchmark.
This technique should be combined with other techniques to yield better results. For doing this testing, it is essential to have skilled and experienced testers.

A significant advantage of this testing technique is that it uncovers the defects in the areas which otherwise remains undetected by other formal testing technique.

State Transition Testing

State Transition Testing is a type of software testing which is performed to check the change in the state of the application under varying input. The condition of input passed is changed and the change in state is observed.
State Transition Testing is basically a black box testing technique that is carried out to observe the behavior of the system or application for different input conditions passed in a sequence. In this type of testing, both positive and negative input values are provided and the behavior of the system is observed.

State Transition Testing is basically used where different system transitions are needed to be tested.



Objectives of State Transition Testing:
The objective of State Transition testing is:

- To test the behavior of the system under varying input.
- To test the dependency on the values in the past.

- To test the change in transition state of the application.
- To test the performance of the system.

Transition States:

- **Change Mode:**
  When this mode is activated then the display mode moves from TIME to DATE.
- **Reset:**
  When the display mode is TIME or DATE, then reset mode sets them to ALTER TIME or ALTER DATE respectively.
- **Time Set:**
  When this mode is activated, display mode changes from ALTER TIME to TIME.
- **Date Set:**
  When this mode is activated, display mode changes from ALTER DATE to DATE.

State Transition Diagram:
State Transition Diagram shows how the state of the system changes on certain inputs.
It has four main components:

1. States
2. Transition
3. Events
4. Actions

Advantages of State Transition Testing:

- State transition testing helps in understanding the behavior of the system.
- State transition testing gives the proper representation of the system behavior.
- State transition testing covers all the conditions.

Disadvantages of State Transition Testing:

- State transition testing can not be performed everywhere.
- State transition testing is not always reliable.

What is State Transition Testing?

State Transition Testing is a black box testing technique in which changes made in input conditions cause state changes or output changes in the Application under Test(AUT). State transition testing helps to analyze behaviour of an application for different input conditions. Testers can provide positive and negative input test values and record the system behavior. It is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.

State Transition Testing Technique is helpful where you need to **test different system transitions**

When to Use State Transition?

- This can be used when a tester is testing the application for a finite set of input values.
- When the tester is trying to test sequence of events that occur in the application under test. I.e., this will allow the tester to test the application behavior for a sequence of input values.
- When the system under test has a dependency on the events/values in the past.

When to Not Rely On State Transition?

- When the testing is not done for sequential input combinations.
- If the testing is to be done for different functionalities like exploratory testing

Four Parts Of State Transition Diagram

There are 4 main components of the State Transition Model as below

1) States that the software might get

2) Transition from one state to another

3) Events that origin a transition like closing a file or withdrawing money

4) Actions that result from a transition (an error message or being given the cash.)

State Transition Diagram and State Transition Table

There are two main ways to represent or design state transition, State transition diagram, and state transition table.

In state transition diagram the states are shown in boxed texts, and the transition is represented by arrows. It is also called State Chart or Graph. It is useful in identifying valid transitions.

In state transition table all the states are listed on the left side, and the events are described on the top. Each cell in the table represents the state of the system after the event has occurred. It is also called State Table. It is useful in identifying invalid transitions.

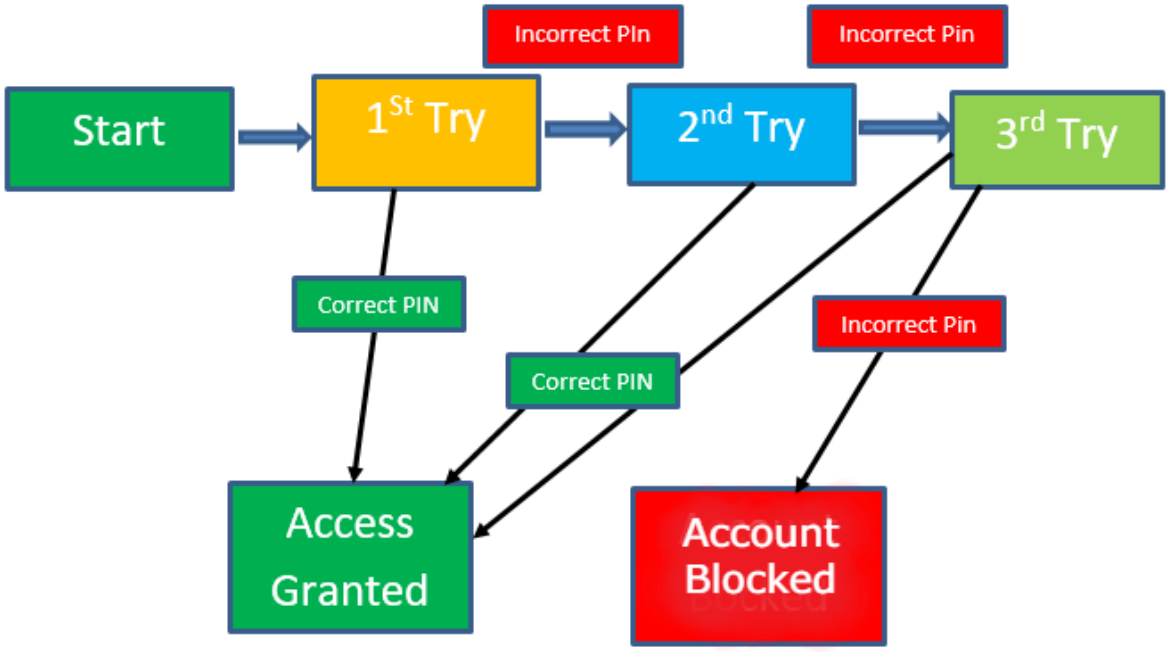
How to Make a State Transition (Examples of a State Transition)

Example 1:

Let's consider an ATM system function where if the user enters the invalid password three times the account will be locked.

In this system, if the user enters a valid password in any of the first three attempts the user will be logged in successfully. If the user enters the invalid password in the first or second try, the user will be asked to re-enter the password. And finally, if the user enters incorrect password 3rd time, the account will be blocked.

State transition diagram



In the diagram whenever the user enters the correct PIN he is moved to Access granted state, and if he enters the wrong password he is moved to next try and if he does the same for the 3rd time the account blocked state is reached.

State Transition Table

	Correct PIN	Incorrect PIN
S1) Start	**S5**	**S2**
S2) 1st attempt	**S5**	**S3**

S3) 2nd attempt	**S5**	**S4**
S4) 3rd attempt	**S5**	**S6**
S5) Access Granted	**–**	**–**
S6) Account blocked	**–**	**–**

In the table when the user enters the correct PIN, state is transitioned to S5 which is Access granted. And if the user enters a wrong password he is moved to next state. If he does the same 3rd time, he will reach the account blocked state.

Decision Table in Testing

What is Decision Table?

Decision tables are used to explain complex logical relationships in many technical areas. With the aid of this testing, the software and its requirements management can be tested very successfully. The decision table provides a tabular view of various combinations of input conditions, which are presented as True(T) and False(F) values. The result may depend on several input circumstances. It also provides a list of conditions and the related activities that must be performed during testing. The decision table technique is one of the frequently used case design strategies for black box testing. This is a methodical approach where different input combinations and their associated system behavior are tabulated. Because of this, it is sometimes referred to as a cause-effect table. This method is used to choose test cases in a methodical manner; it reduces testing time and provides adequate coverage of the software application's testing domain. The decision table method is suitable for functions where there is a logical connection between two or more inputs. This method determines the outcome of different combinations of inputs and is related to the proper combination of inputs. Conditions and actions must be taken into account when designing the test cases using the decision table technique.

Example to Understand Decision Table in Testing

Let's look at an illustration. We are evaluating a web application's login feature. Username and password are the only two fields on the login form. Based on various combinations of the inputs, there are three potential outcomes:

- The user should be given access if their username and password are both accurate.
- An error message should be displayed to the user if either the username or password is incorrect.
- A general error message ought to be displayed to the user if both the username and password are incorrect.

Decision Table:

Situation	Username	Password	Expected Outcome
1	Correct	Correct	Grant Access

2	**Incorrect**	**Correct**	**Invalid Username**
3	**Correct**	**Incorrect**	**Invalid Password**
4	**Incorrect**	**Incorrect**	**Generic Error**

Based on the combination of the inputs—a correct username and a correct password—there are four possible outcomes in this example. There is a predicted consequence for each scenario. This decision table aids in determining the test cases that must be carried out during black box testing in order to cover all potential scenarios and guarantee that the login functionality functions as intended.

Different Parts Decision Table:

The decision table used in software testing is divided into 4 components, which are listed below.

1. **Condition Stubs:** In this first top left area of the decision table, the conditions that are used to choose a certain action or set of actions are listed.
2. **Action Stubs:** All possible actions are included in the decision table's first lower left part (below the condition stub).
3. **Condition Entries:** In the upper right corner of the decision table, values are inserted for the condition entry. Numerous rows and columns that make up the condition entries area of the table are referred to as rules.
4. **Action Entries:** In the lower right corner of the decision table, each item in the action entry has a connected action or a collection of connected actions. "Outputs" are the name given to these values.

Why Decision Table is Important in Software Testing?

A decision table is an excellent tool for managing requirements and testing. The following are some of the factors that make the decision table crucial:

- The test design technique greatly benefits from the use of decision tables.
- It aids testers in investigating the results of various input combinations and other software states that apply business rules.
- It offers a consistent manner of expressing intricate business rules, which is advantageous to both developers and testers.
- It helps the developer do a better job during the development process. Testing every possible combination might not be feasible.
- It is the best option for managing tests and requirements.
- When dealing with intricate business standards, requirements preparation is an organized process.
- Intricate logic is also modeled using it.

What is the scope of the Decision Table in Testing?

When working with complex data and making sure that every possible combination has been taken into account, decision tables can quickly grow to enormous sizes. If you make wise decisions, you can narrow down the list of options in each option to just select the most intriguing and significant ones. Testing using a collapsed decision table is the name of this method. A number of outputs are generated during this process, and redundant criteria that have no influence on the outcome are deleted. In order to aid the tester in conducting testing more effectively, a second layer of analysis is added to the test design. Decision tables are a trustworthy specification-based testing method that can be used in a variety of circumstances. The tabular and graphical representations are simple to understand for non-technical users and all stakeholders. Through instructional examples and real-world situations, project team members can quickly develop a thorough understanding of the topic

at hand. Realizing the usefulness and efficacy of this testing technique may be possible by moving up to the next level of the collapsible decision-making table.

Types of Decision Tables in Testing

Two categories, which are given below, can be used to categorize the decision tables:

1. **Limited Entry:** In limited entry decision tables, the condition elements can only take on binary values.
2. **Extended Entry:** The condition entries of the expanded entry decision table have more than two values. Decision tables with extended entries use many conditions and a variety of outcomes for each condition as opposed to merely true or false.

What is the Applicability of the Decision Table?

- The order in which the rules are applied does not affect the outcome.
- The decision tables can only be used effectively at the unit level.
- After a rule is followed and a course of action is decided upon, another rule must be considered.
- Many applications are not entirely barred by the restrictions.

What are the Advantages of the Decision Table?

The following benefits of employing decision tables in black box testing can be summed up:

1. **Comprehensive Coverage:** Thorough test coverage is achieved by using decision tables, which offer a methodical and structured approach to testing all conceivable combinations of conditions and accompanying actions.
2. **Clarity and Readability:** Decision tables make it simpler for testers to comprehend and confirm the intended behavior of the system by providing a clear and concise representation of complicated business rules.
3. **Test Case Creation:** Decision tables provide as a guide for creating test cases, allowing testers to come up with a clear list of inputs and expected results while lowering the possibility of missing important instances.
4. **Efficiency:** Decision tables enable efficient test design and execution, saving time and effort during the testing process by arranging conditions and actions in a tabular style.
5. **Traceability:** Because decision tables clearly show the relationship between requirements, conditions, and actions, testers may follow the development of a particular test case and confirm that it is in line with the functionality that is intended.
6. **Maintainability:** Because changes to the criteria or actions may be made directly in the table, decision tables are simple to maintain and update, ensuring that the test cases are always current with changing system requirements.

What are the Disadvantages of the Decision Table?

Although decision tables have certain advantages, they also have significant drawbacks in black box testing. These are the main ideas:

1. **Complexity:** As the number of circumstances and actions rises, decision tables may become difficult to handle and complex. They may be challenging to comprehend and sustain as a result.
2. **Scalability:** It becomes difficult to manage the combinations of inputs and outputs as decision tables get larger. Testing every combination is feasible, but it can be time-consuming.
3. **Lack of visibility:** Decision tables do not give extensive information about the inner workings or functioning of the system under test; instead, they simply give a high-level representation of inputs and outputs. The tester’s comprehension of the underlying processes is so constrained.

4. **Limitations in Fault Detection:** Because decision tables concentrate on a small number of input combinations, they may overlook errors or unforeseen events that are not expressly addressed in the table.
5. **Maintenance Burden:** Modifying the decision table may be necessary for any system upgrades or modifications, which can be laborious and error-prone, particularly if the table is large or complex.
6. **Lack of Test Coverage:** Relying entirely on decision tables may lead to insufficient test coverage because they might not take into consideration all scenarios or edge cases, thus causing problems to go undetected.

Levels of Testing

In this section, we are going to understand the various **levels of software testing**.

As we learned in the earlier section of the software testing tutorial that testing any application or software, the test engineer needs to follow multiple testing techniques.

In order to detect an error, we will implement software testing; therefore, all the errors can be removed to find a product with more excellent quality.

What are the levels of Software Testing?

Testing levels are the procedure for finding the missing areas and avoiding overlapping and repetition between the development life cycle stages. We have already seen the various phases such as **Requirement collection, designing, coding testing, deployment, and maintenance** of SDLC (Software Development Life Cycle).

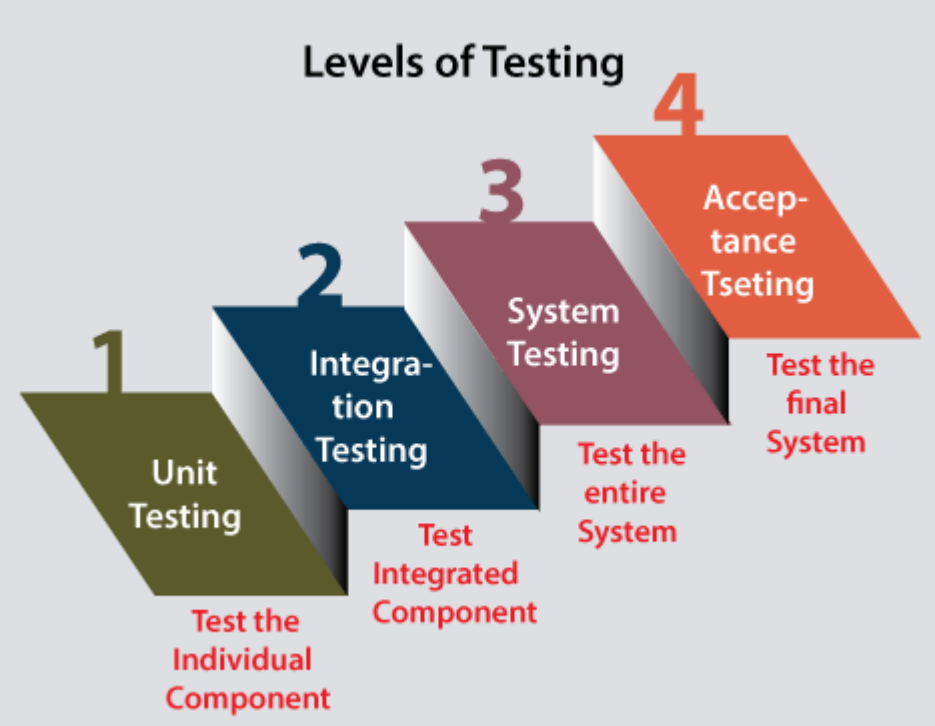
In order to test any application, we need to go through all the above phases of SDLC. Like SDLC, we have multiple levels of testing, which help us maintain the quality of the software.

Different Levels of Testing

The levels of software testing involve the different methodologies, which can be used while we are performing the software testing.

In software testing, we have four different levels of testing, which are as discussed below:

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **Acceptance Testing**



As we can see in the above image that all of these testing levels have a specific objective which specifies the value to the software development lifecycle.

For our better understanding, let's see them one by one:

Level1**: Unit Testing**

Unit testing is the first level of software testing, which is used to test if software modules are satisfying the given requirement or not.

The first level of testing involves **analyzing each unit or an individual component** of the software application.

Unit testing is also the first level of **functional testing**. The primary purpose of executing unit testing is to validate unit components with their performance.

A unit component is an individual function or regulation of the application, or we can say that it is the smallest testable part of the software. The reason of performing the unit testing is to test the correctness of inaccessible code.

Unit testing will help the test engineer and developers in order to understand the base of code that makes them able to change defect causing code quickly. The developers implement the unit.

Level2**: Integration Testing**

The second level of software testing is the **integration testing.** The integration testing process comes after **unit testing**.

It is mainly used to test the **data flow from one module or component to other modules.**

In integration testing, the **test engineer** tests the units or separate components or modules of the software in a group.

The primary purpose of executing the integration testing is to identify the defects at the interaction between integrated components or units.

When each component or module works separately, we need to check the data flow between the dependent modules, and this process is known as **integration testing**.

We only go for the integration testing when the functional testing has been completed successfully on each application module.

In simple words, we can say that **integration testing** aims to evaluate the accuracy of communication among all the modules.

.

Level3: System Testing

The third level of software testing is **system testing**, which is used to test the software's functional and non-functional requirements.

It is **end-to-end testing** where the testing environment is parallel to the production environment. In the third level of software testing, **we will test the application as a whole system.**

To check the end-to-end flow of an application or the software as a user is known as **System testing**.

In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

In simple words, we can say that System testing is a sequence of different types of tests to implement and examine the entire working of an integrated software computer system against requirements.

Level4: Acceptance Testing

The **last and fourth level** of software testing is **acceptance testing**, which is used to evaluate whether a specification or the requirements are met as per its delivery.

The software has passed through three testing levels (**Unit Testing, Integration Testing, System Testing**). Some minor errors can still be identified when the end-user uses the system in the actual scenario.

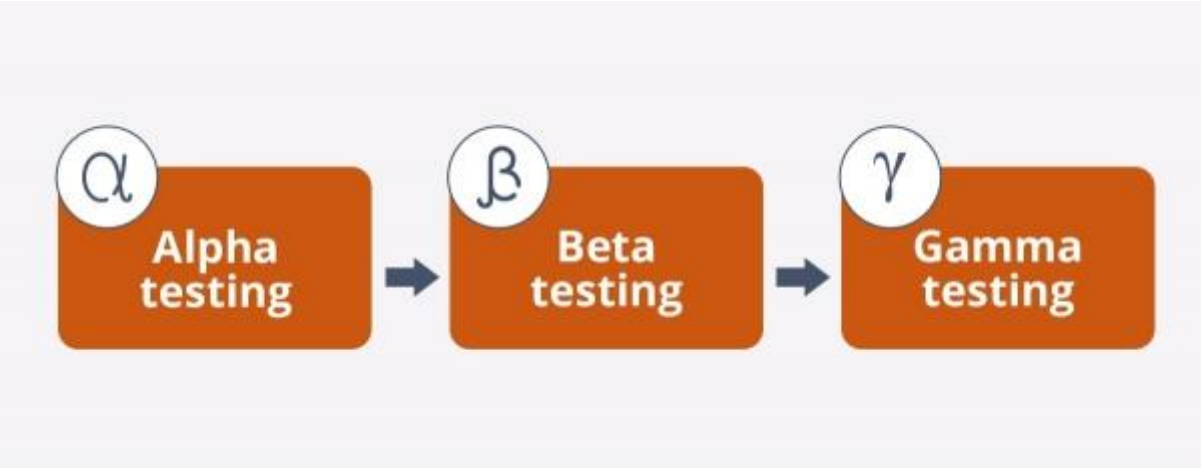
In simple words, we can say that Acceptance testing is the **squeezing of all the testing processes that are previously done.**

The acceptance testing is also known as **User acceptance testing (UAT)** and is done by the customer before accepting the final product.

Usually, UAT is done by the domain expert (customer) for their satisfaction and checks whether the application is working according to given business scenarios and real-time scenarios

Difference Between Alpha, Beta, and Gamma Testing

There are three phases of software testing - alpha, beta, and gamma. They are performed one after another, and together ensure a release of high-quality software.



Alpha Testing

Alpha testing is an internal checking done by the in-house development or QA team, rarely, by the customer himself. Its main purpose is to discover software bugs that were not found before. At the stage of alpha testing, software behavior is verified under real-life conditions by imitating the end-users' actions. It enables us to get fast approval from the customer before proceeding to product delivery.
The alpha phase includes the following testing types: smoke, sanity, integration, systems, usability, UI (user interface), acceptance, regression, and functional testing. If an error is detected, then it is immediately addressed to the development team. Alpha testing helps to discover issues missed at the stage of requirement gathering. The alpha release is the software version that has passed alpha testing. The next stage is beta testing.

Beta Testing

Beta testing can be called pre-release testing. It can be conducted by a limited number of end-users called beta testers before the official product delivery. The main purpose of beta testing is to verify software compatibility with different software and hardware configurations, types of network connection, and to get the users' feedback on software usability and functionality.

There are two types of beta testing:

- **open beta** is available for a large group of end-users or to everyone interested
- **closed beta** is available only to a limited number of users that are selected especially for beta testing.

During beta testing, end users detect and report bugs they have found. All the testing activities are performed outside the organization that has developed the product. Beta checking helps to identify the gaps between the stage of requirements gathering and their implementation. The product version that has passed beta testing is called beta release. After the beta phase comes gamma testing.

Gamma Testing

Gamma testing is the final stage of the testing process conducted before software release. It makes sure that the product is ready for market release according to all the specified requirements. Gamma testing focuses on software security and functionality. But it does not include any in-house QA activities. During gamma testing, the software does not undergo any modifications unless the detected bug is of a high priority and severity.

Only a limited number of users perform gamma testing, and testers do not participate. The checking includes the verification of certain specifications, not the whole product. Feedback received after gamma testing is considered as updates for upcoming software versions. But, because of a limited development cycle, gamma testing is usually skipped.

	Alpha	Beta	Gamma
Why?	validate software in all perspective, ensure readiness for beta testing	get end users' feedback, ensure readiness for release	check software readiness to the specified requirements
When?	at the end of development process	after alpha testing	after beta testing
Who?	in-house development or QA team, customer	a group of real end users	limited number of end users
What get?	bugs, blockers, missing features and others	ideas to improve usability, compatibility, functionality	ideas for updates in upcoming versions
What next?	beta testing	gamma testing	gold release / final release

Bonus: software release life cycle also includes the **pre-alpha stage** that consists of activities done before the QA and testing phase. They are:

- analysis of project requirements
- testing of software requirements
- preparation of a Test Plan

- writing of test cases and test scenarios
- execution of unit testing