What is Software Testing?

Software Testing is a method to assess the functionality of the software program. The process checks whether the actual software matches the expected requirements and ensures the software is bug-free. The purpose of software testing is to identify the errors, faults, or missing requirements in contrast to actual requirements. It mainly aims at measuring the specification, functionality, and performance of a software program or application.

Software testing can be divided into two steps:

1. **Verification:** It refers to the set of tasks that ensure that the software correctly implements a specific function.

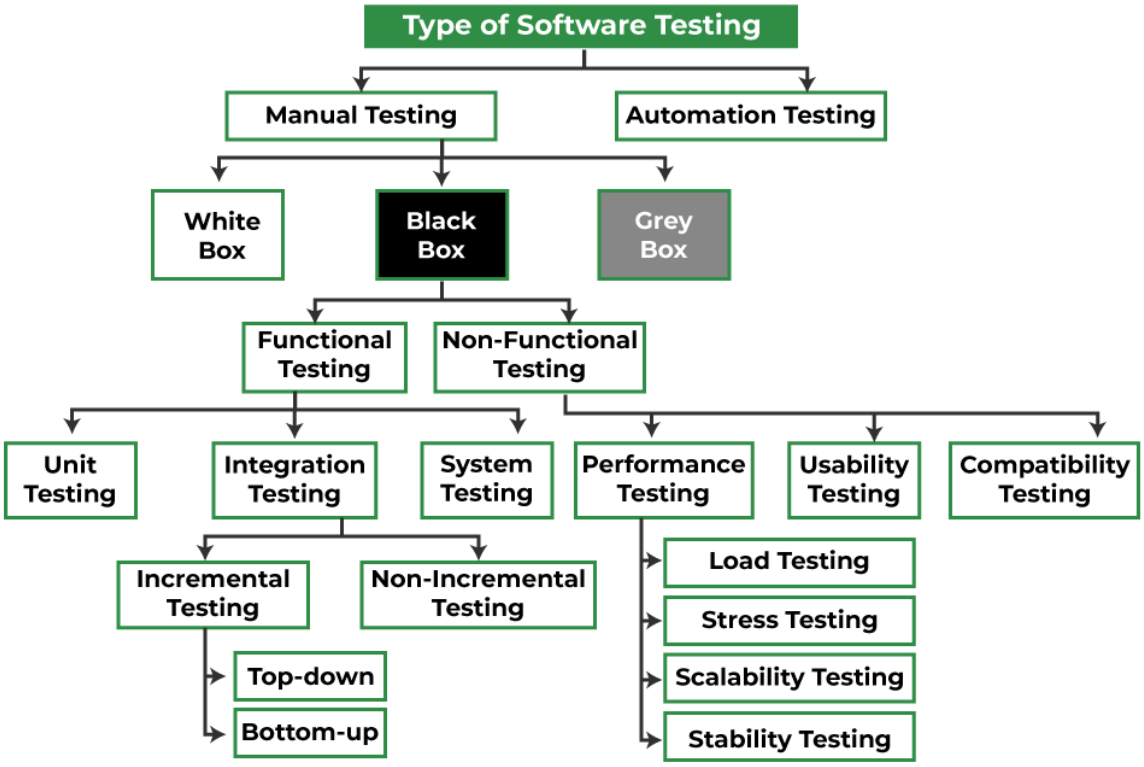
2. **Validation:** It refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

Importance of Software Testing:

- **Defects can be identified early:** Software testing is important because if there are any bugs they can be identified early and can be fixed before the delivery of the software.
- **Improves quality of software:** Software Testing uncovers the defects in the software, and fixing them improves the quality of the software.
- **Increased customer satisfaction:** Software testing ensures reliability, security, and high performance which results in saving time, costs, and customer satisfaction.
- **Helps with scalability:** Software testing type non-functional testing helps to identify the scalability issues and the point where an application might stop working.
- **Saves time and money:** After the application is launched it will be very difficult to trace and resolve the issues, as performing this activity will incur more costs and time. Thus, it is better to conduct software testing at regular intervals during software development.

Different Types Of Software Testing



software testing can be further divided into 2 more ways of testing:

1. **Manual Testing:** Manual testing includes testing software manually, i.e., without using any automation tool or script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behaviour or bug. There are different stages for manual testing such as unit testing, integration testing, system testing, and user acceptance testing. Testers use test plans, test cases, or test scenarios to test software to ensure the completeness of testing. Manual testing also includes exploratory testing, as testers explore the software to identify errors in it.
2. **Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves the automation of a manual process. Automation Testing is used to re-

run the test scenarios quickly and repeatedly, that were performed manually in manual testing.
Apart from regression testing, automation testing is also used to test the application from a load, performance, and stress point of view. It increases the test coverage, improves accuracy, and saves time and money when compared to manual testing.

Internal and external View of Testing

The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing. Software testing techniques can be majorly classified into two categories:

1. **Black Box Testing:** The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without any concern with the internal logical structure of the software is known as black-box testing.

2. **White-Box Testing:** The technique of testing in which the tester is aware of the internal workings of the product, has access to its source code, and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

Differences between Black Box Testing vs White Box Testing:

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred to as outer or external software testing.	It is the inner or the internal software testing.

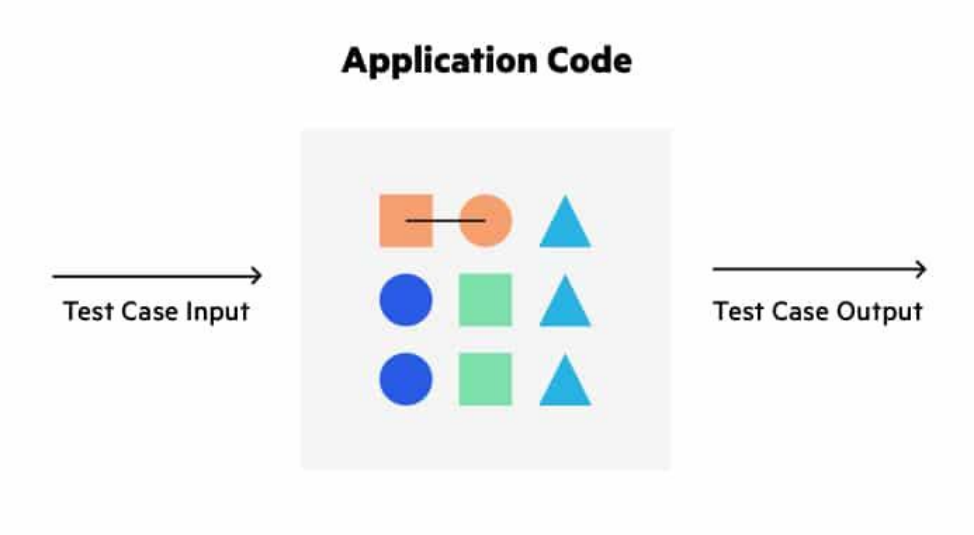
Black Box Testing	White Box Testing
It is a functional test of the software.	It is a structural test of the software.
This testing can be initiated based on the requirement specifications document.	This type of testing of software is started after a detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: Search something on google by using keywords	**Example:** By input to check and verify loops
Black-box test design techniques- • Decision table testing • All-pairs testing • Equivalence partitioning • Error guessing	**White-box test design techniques-** • Control flow testing • Data flow testing • Branch testing
Types of Black Box Testing: • Functional Testing • Non-functional testing • Regression Testing	**Types of White Box Testing:** • Path Testing • Loop Testing • Condition testing
It is less exhaustive as compared to white box testing.	It is comparatively more exhaustive than black box testing.

What are different levels of software testing?
Software level testing can be majorly classified into 4 levels:

1. **Unit Testing:** A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing:** A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.
3. **System Testing:** A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing:** A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

White box Testing

White box testing techniques analyze the internal structures the used data structures, internal design, code structure, and the working of the software rather than just the functionality as in black box testing. It is also called **glass box testing** or clear box testing or structural testing. White Box Testing is also known as transparent testing or open box testing.

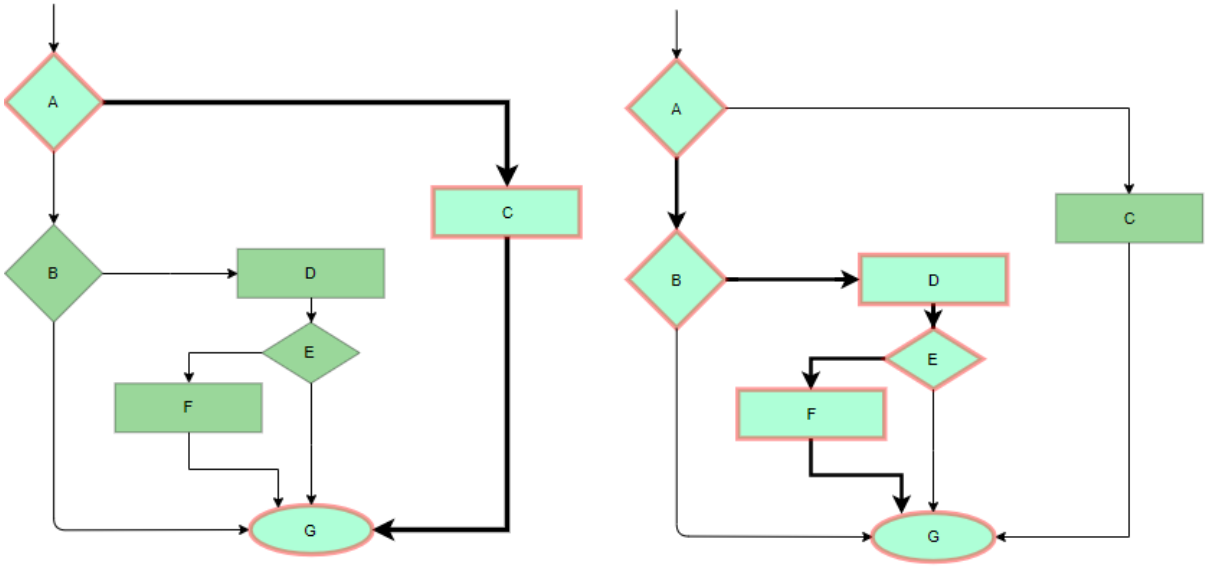


White box testing is a software testing technique that involves testing the internal structure and workings of a software application. The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level.

White box testing is also known as structural testing or code-based testing, and it is used to test the software's internal logic, flow, and structure. The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements.

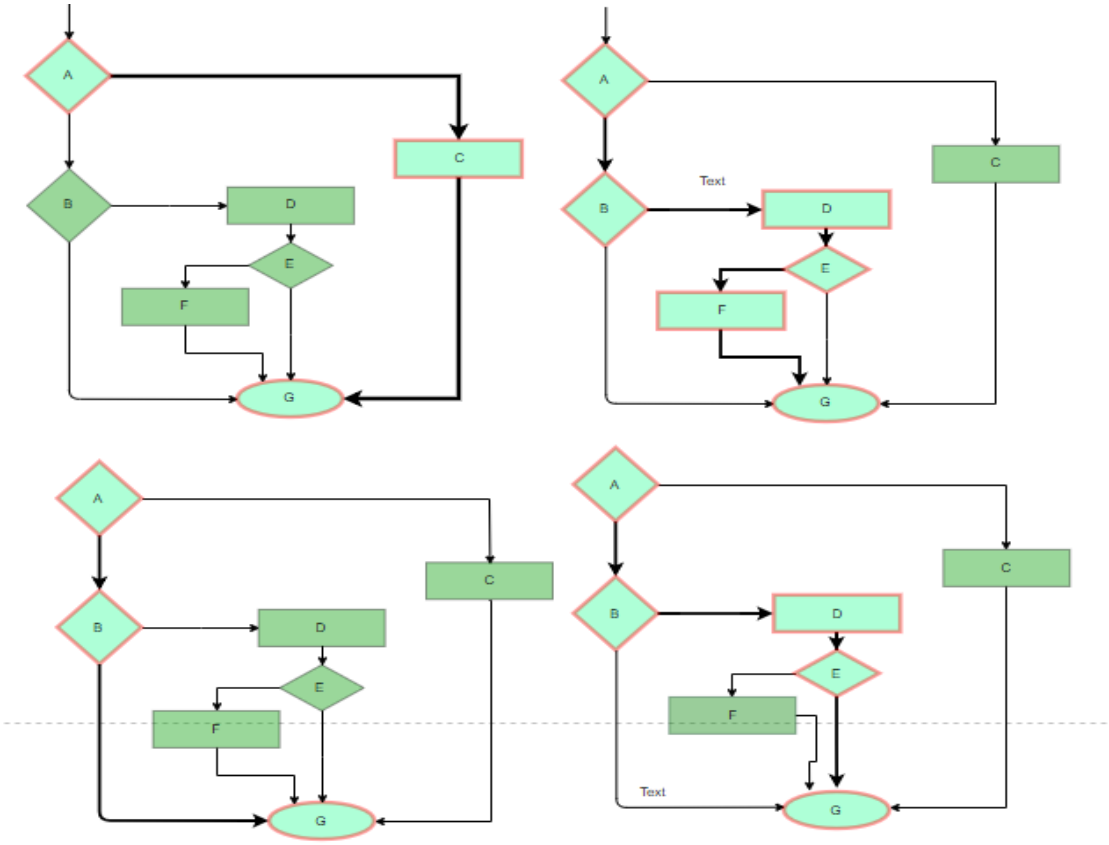
Testing techniques:

- **Statement coverage:** In this technique, the aim is to traverse all statements at least once. Hence, each line of code is tested. In the case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



Statement Coverage Example

- **Branch Coverage:** In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.



4 test cases are required such that all branches of all decisions are covered, i.e, all edges of the flowchart are covered

- **Condition Coverage:** In this technique, all individual conditions must be covered as shown in the following example:
    1. INPUT A, B
    2. C = A + B
    3. IF C>100
    4. PRINT “Its done”
    5. ELSE
    6. PRINT “It’s Not Done”

Since the goal of white box testing is to verify and cross-check all the different loops, branches and decision statements, so to exercise white box testing in the code given above, the two test cases would be –

- A= 33, B=45
- A=40, B=70

For the first test case, A=33, B=45; C becomes 78, due to which it will skip the 4th line in the pseudo-code, since C < 100 and will directly print the 6th line, i.e ITS Not Done.

Now, for the second test case, A=40, B=70; C becomes 110, which means that C > 100 and therefore it will print the 4th line and the program will be stopped.

These test cases will ensure that each line of the code is traversed at least once and will verify for both true and false conditions.

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

Tools required for White box Testing:

- PyUnit
- Sqlmap
- Nmap
- Parasoft Jtest
- Nunit
- VeraUnit
- CppUnit
- Bugzilla

Features of white box testing:

1. **Code coverage analysis:** White box testing helps to analyze the code coverage of an application, which helps to identify the areas of the code that are not being tested.
2. **Access to the source code:** White box testing requires access to the application's source code, which makes it possible to test individual functions, methods, and modules.
3. **Knowledge of programming languages:** Testers performing white box testing must have knowledge of programming languages like Java, C++, Python, and PHP to understand the code structure and write tests.
4. **Identifying logical errors:** White box testing helps to identify logical errors in the code, such as infinite loops or incorrect conditional statements.
5. **Integration testing:** White box testing is useful for integration testing, as it allows testers to verify that the different components of an application are working together as expected.

6. **Unit testing:** White box testing is also used for unit testing, which involves testing individual units of code to ensure that they are working correctly.
7. **Optimization of code:** White box testing can help to optimize the code by identifying any performance issues, redundant code, or other areas that can be improved.
8. **Security testing:** White box testing can also be used for security testing, as it allows testers to identify any vulnerabilities in the application's code.

Advantages:

1. White box testing is thorough as the entire code and structures are tested.
2. It results in the optimization of code removing errors and helps in removing extra lines of code.
3. It can start at an earlier stage as it doesn't require any interface as in the case of black box testing.
4. Easy to automate.
5. White box testing can be easily started in Software Development Life Cycle.
6. Easy Code Optimization.

- Testers can identify defects that cannot be detected through other testing techniques.

- Testers can create more comprehensive and effective test cases that cover all code paths.

- Testers can ensure that the code meets coding standards and is optimized for performance.

Disadvantages:

1. It is very expensive.
2. Redesigning code and rewriting code needs test cases to be written again.
3. Testers are required to have in-depth knowledge of the code and programming language as opposed to black-box testing.
4. Missing functionalities cannot be detected as the code that exists is tested.
5. Very complex and at times not realistic.
6. Much more chances of Errors in production.

Basis Path Testing in Software Testing

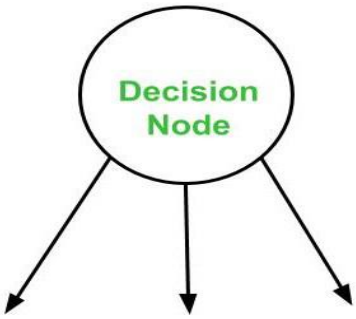
Prerequisite – Path Testing **Basis Path Testing** is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition, 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 :

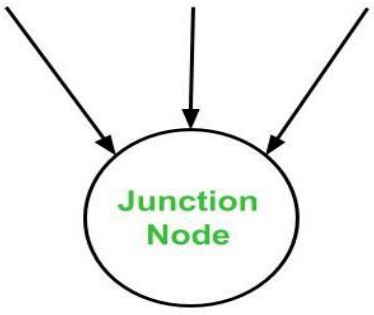
1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths

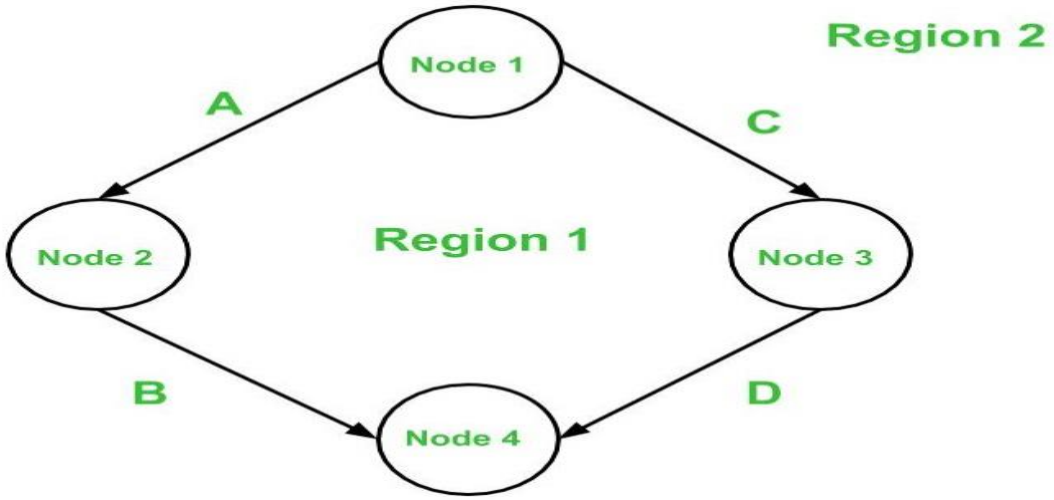
Let's understand each step one by one.

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.).

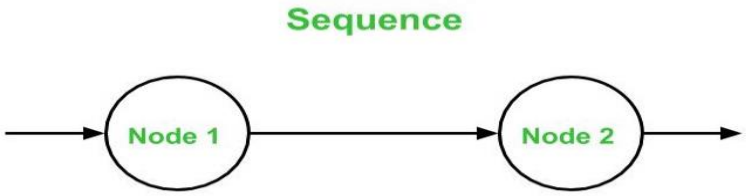






Below are the **notations** used while constructing a flow graph :

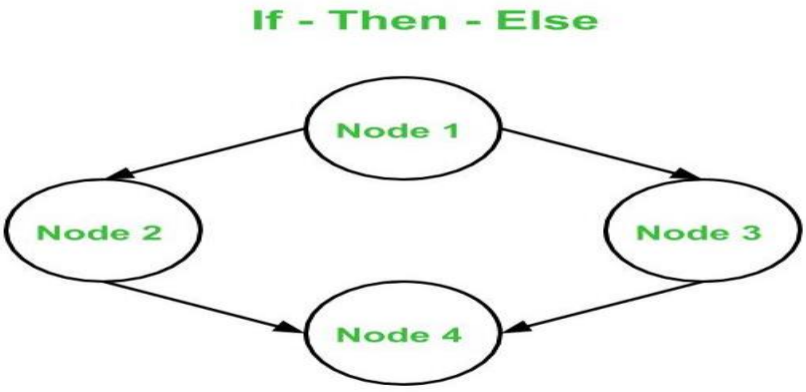
- **Sequential Statements**



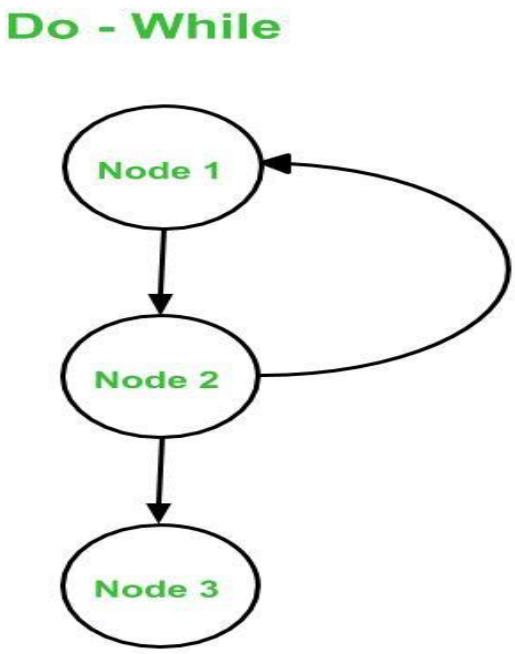
– **If –**

Then – Else

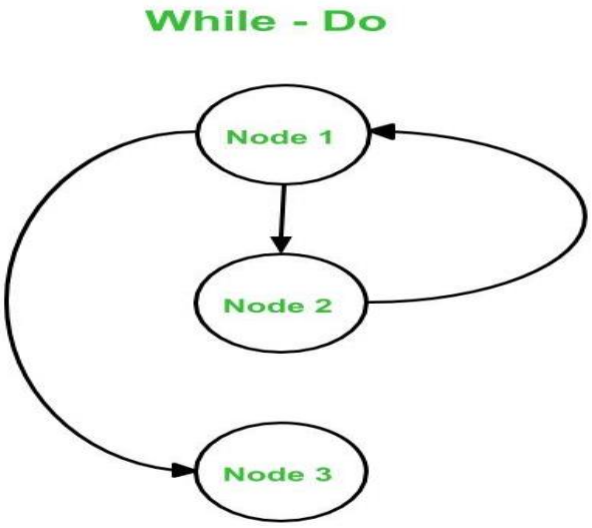
–



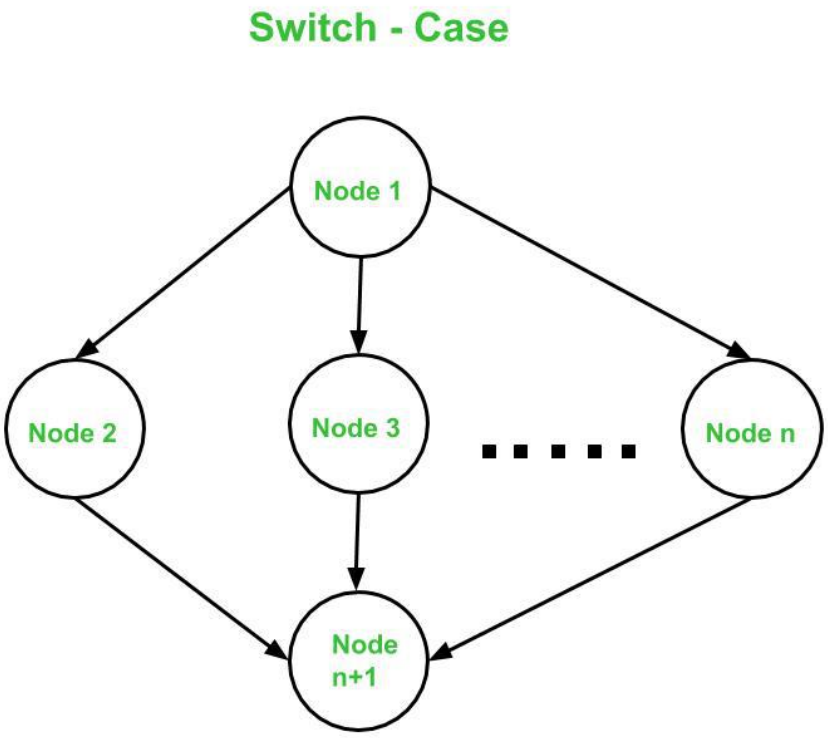
- **Do – While**
  –



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.

Control Structure Testing

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. Condition Testing : Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on. The common types of logical conditions that are tested using condition testing are-

1. A relation expression, like E1 op E2 where ‘E1’ and ‘E2’ are arithmetic expressions and ‘OP’ is an operator.

2. A simple condition like any relational expression preceded by a NOT (~) operator. For example, (~E1) where ‘E1’ is an arithmetic expression and ‘a’ denotes NOT operator.
3. A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis. For example, (E1 & E2)|(E2 & E3) where E1, E2, E3 denote arithmetic expression and ‘&’ and ‘|’ denote AND or OR operators.
4. A Boolean expression consists of operands and a Boolean operator like ‘AND’, OR, NOT. For example, ‘A|B’ is a Boolean expression where ‘A’ and ‘B’ denote operands and | denotes OR operator.

2. Data Flow Testing : The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program. The data flow test approach is depicted as follows suppose each statement in a program is assigned a unique statement number and that theme function cannot modify its parameters or global variables. For example, with S as its statement number.

DEF (S) = {X | Statement S has a definition of X}

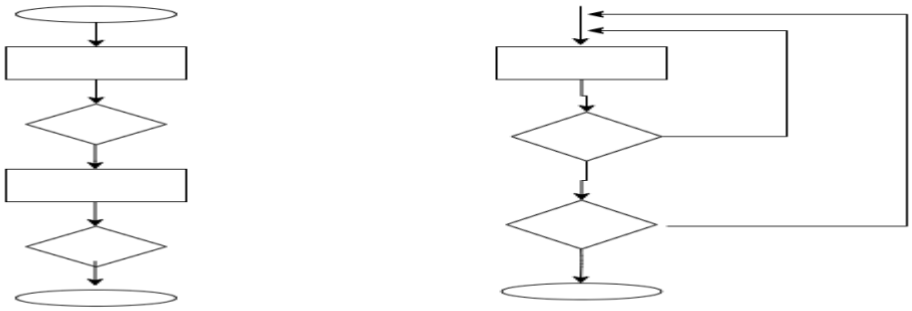
USE (S) = {X | Statement S has a use of X}

If statement S is an if loop statement, them its DEF set is empty and its USE set depends on the state of statement S. The definition of the variable X at statement S is called the line of statement S’ if the statement is any way from S to statement S’ then there is no other definition of X. A definition use (DU) chain of variable X has the form [X, S, S’], where S and S’ denote statement numbers, X is in DEF(S) and USE(S’), and the definition of X in statement S is line at statement S’. A simple data flow test approach requires that each DU chain be covered at least once. This approach is known as the DU test approach. The DU testing does not ensure coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare cases such as then in which the other construct does not have any certainty of any variable in its later part and the other part is not present. Data flow testing strategies are appropriate for choosing test paths of a program containing nested if and loop statements.

3. Loop Testing : Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction. Following are the types of loops.

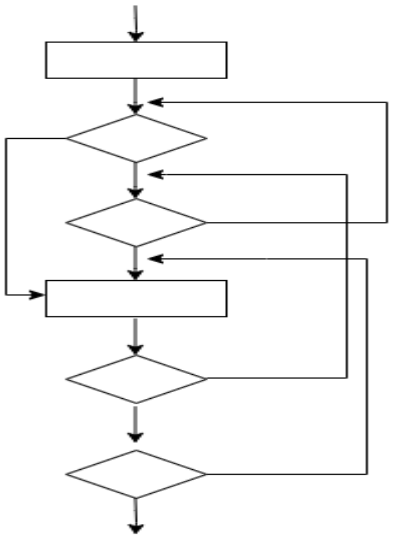
1. **Simple Loop –** The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n.
    1. Skip the entire loop.
    2. Traverse the loop only once.
    3. Traverse the loop two times.
    4. Make p passes through the loop where p<n.
    5. Traverse the loop n-1, n, n+1 times.
2. **Concatenated Loops –** If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested

loops.



Concatenated Loops

3. **Nested Loops –** Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases. The following steps for testing nested loops are as follows-
    1. Start with inner loop. set all other loops to minimum values.
    2. Conduct simple loop testing on inner loop.
    3. Work outwards.
    4. Continue until all loops tested.
4. **Unstructured loops –** This type of loops should be redesigned, whenever possible, to reflect the use of unstructured the structured programming



constructs.

Unstructured Loops

Black Box Testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

The above Black-Box can be any software system you want to test. For Example, an operating system like Windows, a website like Google, a database like Oracle or even your own custom application. Under Black Box Testing, you can test these applications by just focusing on the inputs and outputs without knowing their internal code implementation.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

It does not require programming knowledge of the software. All test cases are designed by considering the input and output of a particular function.A tester knows about the definite output of a particular input, but not about how the result is arising. There are various techniques used in black box testing for testing like decision table technique, boundary value analysis technique, state transition, All-pair testing, cause-effect graph technique, equivalence partitioning technique, error guessing technique, use case technique and user story technique. All these techniques have been explained in detail within the tutorial.

Test cases

Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values

to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones –

- **Functional testing** – This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** – Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

How to do BlackBox Testing in Software Engineering

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Tools used for Black Box Testing:

Tools used for Black box testing largely depends on the type of black box testing you are doing.

- For Functional/ Regression Tests you can use – QTP, Selenium
- For Non-Functional Tests, you can use – LoadRunner, Jmeter

Debugging

Debugging is the process of identifying and resolving errors, or bugs, in a software system. It is an important aspect of software engineering because bugs can cause a software system to malfunction, and can lead to poor performance or incorrect results. Debugging can be a time-consuming and complex task, but it is essential for ensuring that a software system is functioning correctly.
There are several common methods and techniques used in debugging, including:

1. **Code Inspection:** This involves manually reviewing the source code of a software system to identify potential bugs or errors.
2. **Debugging Tools:** There are various tools available for debugging such as debuggers, trace tools, and profilers that can be used to identify and resolve bugs.
3. **Unit Testing:** This involves testing individual units or components of a software system to identify bugs or errors.
4. **Integration Testing:** This involves testing the interactions between different components of a software system to identify bugs or errors.
5. **System Testing:** This involves testing the entire software system to identify bugs or errors.
6. **Monitoring**: This involves monitoring a software system for unusual behavior or performance issues that can indicate the presence of bugs or errors.
7. **Logging**: This involves recording events and messages related to the software system, which can be used to identify bugs or errors.

It is important to note that debugging is an iterative process, and it may take multiple attempts to identify and resolve all bugs in a software system. Additionally, it is important to have a well-defined process in place for reporting and tracking bugs, so that they can be effectively managed and resolved.

In summary, debugging is an important aspect of software engineering, it's the process of identifying and resolving errors, or bugs, in a software system. There are several common methods and techniques used in debugging, including code inspection, debugging tools, unit testing, integration testing, system testing, monitoring, and logging. It is an iterative process that may take multiple attempts to identify and resolve all bugs in a software system.

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing, and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

A better approach is to run the program within a debugger, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of breakpoints within the code. When the program is executed within

the debugger, it stops at each breakpoint. Many IDEs, such as Visual C++ and C-Builder provide built-in debuggers.

Debugging Process: Steps involved in debugging are:

- Problem identification and report preparation.
- Assigning the report to the software engineer defect to verify that it is genuine.
- Defect Analysis using modeling, documentation, finding and testing candidate flaws, etc.
- Defect Resolution by making required changes to the system.
- Validation of corrections.

The debugging process will always have one of two outcomes :

1. The cause will be found and corrected.
2. The cause will not be found.

Later, the person performing debugging may suspect a cause, design a test case to help validate that suspicion and work toward error correction in an iterative fashion.

During debugging, we encounter errors that range from mildly annoying to catastrophic. As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches/Strategies:

1. **Brute Force:** Study the system for a larger duration in order to understand the system. It helps the debugger to construct different representations of systems to be debugged depending on the need. A study of the system is also done actively to find recent changes made to the software.
2. **Backtracking:** Backward analysis of the problem which involves tracing the program backward from the location of the failure message in order to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.
3. **Forward analysis** of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused on to find the defect.
4. **Using past experience** with the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.
5. **Cause elimination:** it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
6. **Static analysis:** Analyzing the code without executing it to identify potential bugs or errors. This approach involves analyzing code syntax, data flow, and control flow.
7. **Dynamic analysis:** Executing the code and analyzing its behavior at runtime to identify errors or bugs. This approach involves techniques like runtime debugging and profiling.
8. **Collaborative debugging:** Involves multiple developers working together to debug a system. This approach is helpful in situations where multiple modules or components are involved, and the root cause of the error is not clear.
9. **Logging and Tracing:** Using logging and tracing tools to identify the sequence of events leading up to the error. This approach involves collecting and analyzing logs and traces generated by the system during its execution.

10. **Automated Debugging:** The use of automated tools and techniques to assist in the debugging process. These tools can include static and dynamic analysis tools, as well as tools that use machine learning and artificial intelligence to identify errors and suggest fixes.

Debugging Tools:

Debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command-line interfaces. Examples of automated debugging tools include code-based tracers, profilers, interpreters, etc. Some of the widely used debuggers are:

- Radare2
- WinDbg
- Valgrind

Difference Between Debugging and Testing:

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing unit testing, integration testing, alpha, and beta testing, etc. Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

Advantages of Debugging:

Several advantages of debugging in software engineering:

1. **Improved system quality:** By identifying and resolving bugs, a software system can be made more reliable and efficient, resulting in improved overall quality.
2. **Reduced system downtime:** By identifying and resolving bugs, a software system can be made more stable and less likely to experience downtime, which can result in improved availability for users.
3. **Increased user satisfaction**: By identifying and resolving bugs, a software system can be made more user-friendly and better able to meet the needs of users, which can result in increased satisfaction.
4. **Reduced development costs:** By identifying and resolving bugs early in the development process, it can save time and resources that would otherwise be spent on fixing bugs later in the development process or after the system has been deployed.
5. **Increased security**: By identifying and resolving bugs that could be exploited by attackers, a software system can be made more secure, reducing the risk of security breaches.
6. **Facilitates change:** With debugging, it becomes easy to make changes to the software as it becomes easy to identify and fix bugs that would have been caused by the changes.
7. **Better understanding of the system:** Debugging can help developers gain a better understanding of how a software system works, and how different components of the system interact with one another.

8. **Facilitates testing:** By identifying and resolving bugs, it makes it easier to test the software and ensure that it meets the requirements and specifications.

In summary, debugging is an important aspect of software engineering as it helps to improve system quality, reduce system downtime, increase user satisfaction, reduce development costs, increase security, facilitate change, a better understanding of the system, and facilitate testing.

Disadvantages of Debugging:

While debugging is an important aspect of software engineering, there are also some disadvantages to consider:

1. **Time-consuming:** Debugging can be a time-consuming process, especially if the bug is difficult to find or reproduce. This can cause delays in the development process and add to the overall cost of the project.
2. **Requires specialized skills:** Debugging can be a complex task that requires specialized skills and knowledge. This can be a challenge for developers who are not familiar with the tools and techniques used in debugging.
3. **Can be difficult to reproduce:** Some bugs may be difficult to reproduce, which can make it challenging to identify and resolve them.
4. **Can be difficult to diagnose:** Some bugs may be caused by interactions between different components of a software system, which can make it challenging to identify the root cause of the problem.
5. **Can be difficult to fix:** Some bugs may be caused by fundamental design flaws or architecture issues, which can be difficult or impossible to fix without significant changes to the software system.
6. **Limited insight**: In some cases, debugging tools can only provide limited insight into the problem and may not provide enough information to identify the root cause of the problem.
7. **Can be expensive:** Debugging can be an expensive process, especially if it requires additional resources such as specialized debugging tools or additional development time.

In summary, debugging is an important aspect of software engineering but it also has some disadvantages, it can be time-consuming, requires specialized skills, can be difficult to reproduce, diagnose and fix, may have limited insight, and can be expensive.