UNIT I – FUNDAMENTALS

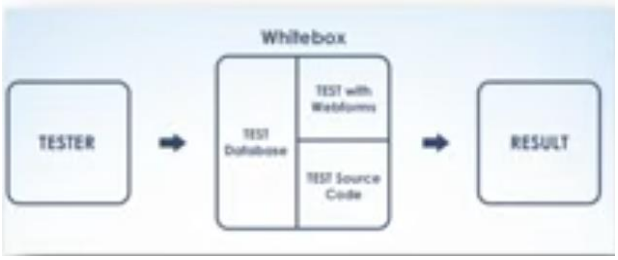
Principles of Testing – Phases of Software Project – Quality Assurance and Control – Verification and Validation - White Box Testing: Static Testing – Structural Testing – Challenges.

1.5 White Box Testing

White box testing is testing based on analysis of internal logic (design, code, etc.). (But expected results still come from requirements.). It is also known as structural testing.

White-box testing concerns techniques for designing tests; it is not a level of testing. White-box testing techniques apply primarily to lower levels of testing (e.g., unit and component).

It is also called clear box testing,open box testing,transparent box testing,code based testing,and glass box testing.



The major testing focuses on

- Program structures.
- Program statements and branches
- Various kinds of program paths
- Program internal logic and data structures.
- Program internal behaviors and states.
- Logic coverage.
  - **Statement:**

Statement Coverage:

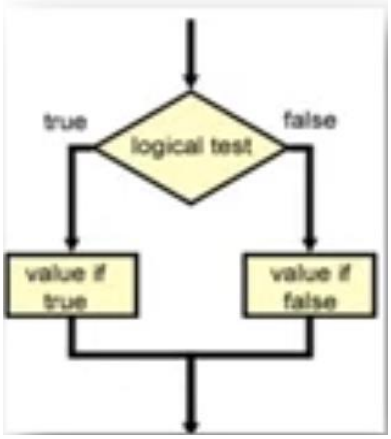
Statement coverage is a testing technique in which tester traverse all statements in code.

It ensure that each statement of the code is executed at least once.

Hence each line of the code is verified

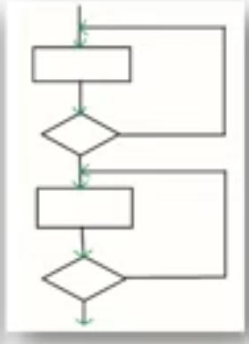
```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

  - **Branch:** each branch traversed (and every entry point taken) at least once.
  - **Condition:** each condition True at least once and False at least once.

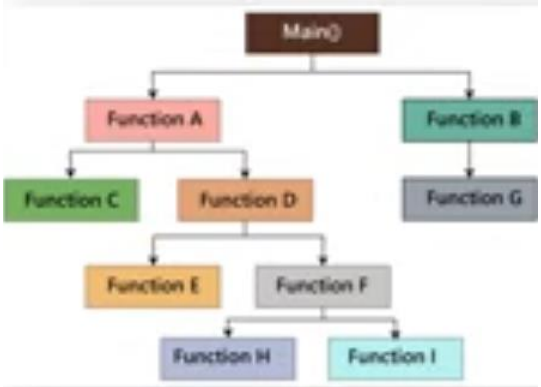


- **Branch/Condition:** both branch and condition coverage achieved.
- Compound Condition: all combinations of condition values at every branch statement covered (and every entry point taken).

```
while(condition 1)
 {
  statement(s);
 }
while(condition 2)
{
 statement(s);
}
```



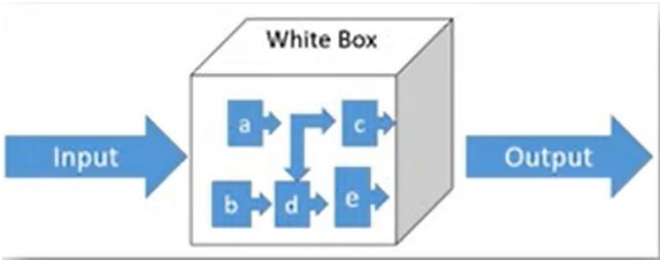
- **Path:** all program paths traversed at least once. Testing is dependent on the program's Control or flow structure.
  All possible paths are defined with entry to the exist point in the system.



- Dataflow coverage.
- Path conditions and symbolic evaluation.
- Other white-box testing strategies (e.g., "fault-based testing").

WHITE BOX TESTING

White box is a testing methodology to test the internal structures and working of software. White box testing also known as structural testing is testing based on analysis of internal logic (design, code, etc.).



Test Model: Control program chart (graph)
Test case design: Various white-box testing methods generate test cases based on a given control program graph for a program.

The goal of white box testing is to:

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to assure their validity.
- Exercise all data define and use paths.

Static Testing

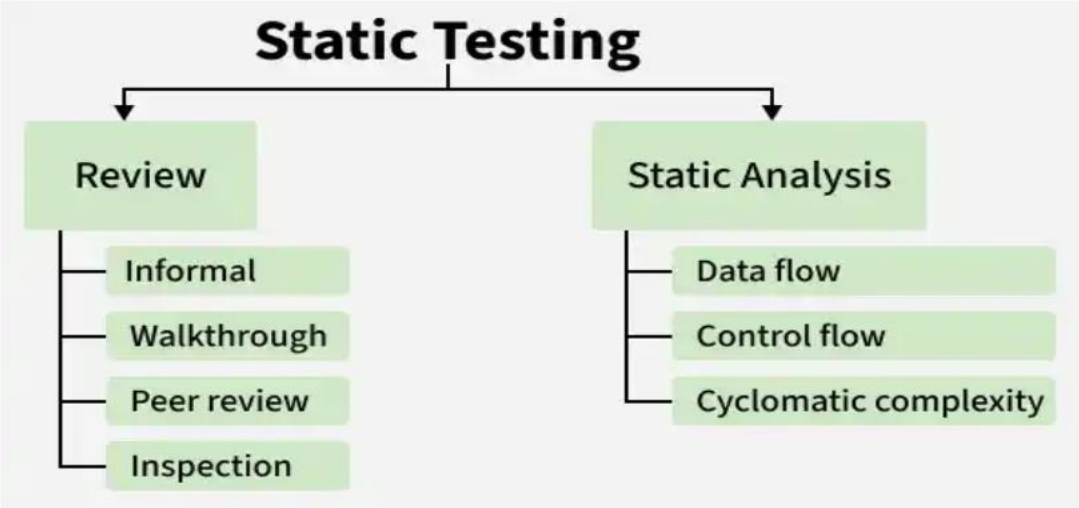
What is Static Testing?

Static Testing, a software testing technique in which the software is tested without executing the code. It is a type of software testing where the application or code is reviewed without executing it. It is mainly used to detect errors in requirements, design documents, and source code at an early stage. This helps in improving software quality before actual execution begins.

- It does not involve running the program or executing code.
- It helps in identifying defects early in the Software Development Life Cycle (SDLC).
- It includes techniques like reviews, walkthroughs, and inspections.

Types of Static Testing

There are mainly two types of techniques used in Static Testing:



Types of Static Testing

Review

A **Review** is a manual static testing technique in which software artifacts such as requirements, design documents, and source code are examined without executing the program. Its main objective is to identify defects early, improve quality, and ensure compliance with standards.

- **Informal Review(Desk Checking):** A casual evaluation of documents through discussions and feedback to identify defects.
- **Walkthrough:** The author presents the document step-by-step to team members to improve understanding and uncover issues.
- **Peer Review:** Colleagues examine each other's work to detect errors and suggest improvements.
- **Inspection(Formal Inspection):** A structured and formal review method that uses predefined roles, procedures, and checklists to identify defects in requirements, design, or code.

Key Characteristics of Formal Inspection

- Driven completely by strict rules, checklists, and defined procedures.
- Enforces explicit entry and exit criteria to control the process quality.
- Led by a trained neutral moderator, never by the document's author.
- Focuses strictly on finding defects and gathering data, not on fixing problems during the meeting.

A formal inspection involves a dedicated group of peers (typically 3 to 8 participants) with distinct responsibilities:

- **Moderator (Facilitator):** Leads the process, schedules the sessions, runs the meeting, and ensures exit criteria are met.
- **Author:** The person who created the document or code under review. Their primary task is to explain anomalies when asked and learn from errors.
- **Reader:** Reads the document aloud or leads the team through the material item by item during the meeting.
- **Recorder (Scribe):** Documents every defect, anomaly, and suggestion in an official issue log.

- **Inspectors (Reviewers):** Team members who thoroughly evaluate the document from different perspectives to catch errors.

Static Analysis

Static Analysis is an automated technique that evaluates source code without executing it. Specialized tools are used to detect defects, enforce coding standards, and improve software quality during development.

- **Data Flow Analysis:** Analyzes how data is created, modified, and used throughout the program.
- **Control Flow Analysis:** Examines the sequence and logic of program execution paths.
- **Cyclomatic Complexity:** Calculates the number of independent execution paths in the code, helping assess complexity and testing requirements.

Static Testing Process

Static Testing is a software testing technique used to evaluate software documents, designs, and source code without executing the program. Its primary goal is to identify defects at an early stage of the Software Development Life Cycle (SDLC).

- **Requirements Review:** Requirement specifications are examined to ensure they are accurate, complete, consistent, and understandable before development starts.
- **Design Evaluation:** Design documents are reviewed to verify that the system architecture and design correctly satisfy the specified requirements.
- **Code Examination:** Source code is analyzed without running it to identify programming mistakes, coding standard violations, and potential defects.
- **Walkthrough:** The developer or author presents the document or code to team members, who provide suggestions and help discover issues through discussion.
- **Inspection:** A structured and formal review method in which reviewers use predefined checklists to systematically identify defects in documents or code.
- **Static Analysis Using Tools:** Specialized software tools automatically scan source code to detect bugs, security vulnerabilities, and coding standard violations without executing the application.

Artifacts Reviewed in Static Testing

Static testing is applied to various software documents and artifacts:

- Requirements documents (BRD, SRS)
- Design documents
- Source code
- Test cases
- Use cases and prototypes
- Traceability matrix
- Test scripts and performance documents

Tools Used in Static Testing

- **SonarQube:** Detects bugs, code smells, and security issues in source code.
- **Checkstyle:** Checks Java code for coding standard and formatting violations.
- **PMD:** Finds common coding problems such as unused variables and empty blocks.
- **ESLint:** Identifies errors and enforces best practices in JavaScript code.
- **SpotBugs (FindBugs):** Analyzes Java bytecode to detect potential bugs.
- **Fortify Static Code Analyzer:** Identifies security vulnerabilities in application source code

Need for Static Testing

- **Complex Software:** Helps identify defects early in large and complex applications.

- **High Cost of Dynamic Testing:** Reduces testing costs by finding issues before execution.
- **Time Constraints:** Saves time by detecting defects in the early stages of development.
- **Improved Productivity:** Minimizes rework and enhances overall development efficiency.

Advantages of Static Testing

- Identifies defects early, reducing effort, time, and cost of fixing them.
- Detects common coding errors such as syntax errors and null pointer issues.
- Improves code structure, maintainability, and overall quality.
- Provides quick feedback throughout the software development process.
- Helps locate defects accurately and efficiently.

Disadvantages of Static Testing

- Cannot detect defects that occur only during program execution, such as memory leaks and performance issues.
- Effectiveness depends on the experience and skills of the reviewer.
- Can be time-consuming for large and complex software projects.
- Requires significant human involvement through manual reviews and inspections.

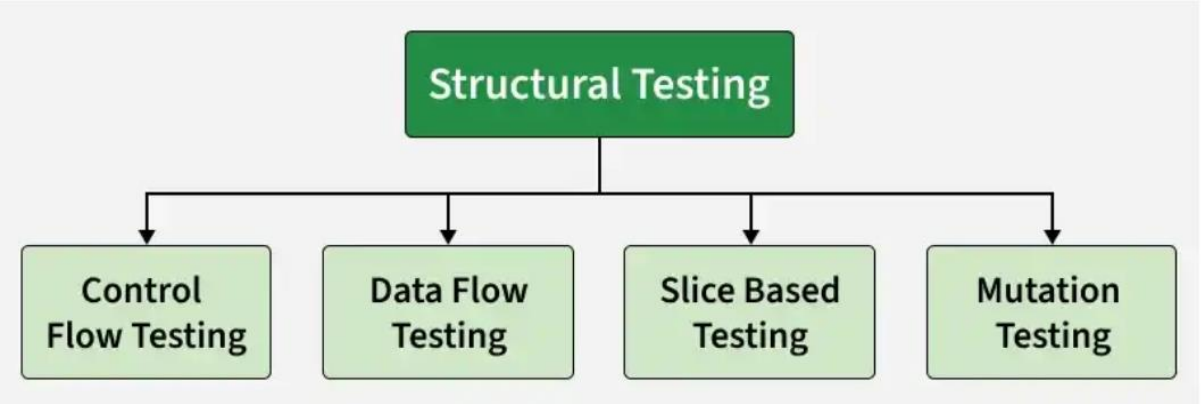
Structural Testing

Structural Testing is a **white-box testing method** that examines the internal structure, logic, and code flow of a software application. It requires knowledge of the program's source code and aims to verify that all code components function correctly.

- Test cases are designed based on the program's internal code structure rather than functional requirements.
- Emphasizes coverage measures such as statement coverage, branch coverage, and path coverage.
- Assists in identifying dead code, unreachable statements, and hidden defects during development.

Types of Structural Testing

There are different types of Structural Testing:



Control Flow Testing

Control Flow Testing is a white-box testing technique that designs test cases based on the program's control flow and execution paths. It focuses on verifying the logical structure, decisions, loops, and flow of control within the code to ensure correct program behavior. As a structural testing method, it uses the program's design and source code to identify and test different execution paths.

Control flow testing helps detect logical errors, unreachable code, and incorrect program flow. For example, in an online banking system, it ensures that all decision branches and loops are thoroughly tested, reducing the risk of logical errors that could result in financial losses.

Types of Control Flow Testing Coverage:

Statement Coverage: Ensures that every executable statement in the program is executed at least once during testing.Statement coverage is a foundational white-box testing technique designed to ensure that every executable statement in the source code is executed at least once during testing. It is a vital code metric used by developers and automation engineers to evaluate the thoroughness of a test suite and discover areas of unexecuted code.To evaluate statement coverage, the metric calculates the percentage of code lines triggered by the test runner using the following standard equation:

$$\text{Statement Coverage} = \left(\frac{\text{Number of Executed Statements}}{\text{Total Number of Executable Statements}} \right) \times 100\%$$

Example:

1. Read total_amount

2. if (total_amount > 100) {

3. discount = total_amount * 0.1

4. total_amount = total_amount - discount

5. }

6. Print total_amount

Scenario A (Partial Coverage): A test case with total_amount = 50 is evaluated. Lines 1, 2, and 6 will execute. Lines 3 and 4 are skipped because the conditional expression returns false.

- Executed Statements: 3 (Lines 1, 2, 6)
- Coverage: (3 / 5) × 100% = 60%

Scenario B (100% Coverage): A test case with total_amount = 150 is evaluated. The conditional expression returns true, driving execution through every single step.

- Executed Statements: 5 (Lines 1, 2, 3, 4, 6)
- Coverage: (5 / 5) × 100% = 100%

Branch Coverage: Ensures that every decision branch (true and false outcomes) is tested at least once.Branch coverage (also known as decision coverage) is a structural white box testing metric that measures the percentage of decision outcomes (such as the True and False paths of an if statement) that are executed during testing. The primary goal is to ensure that every possible path originating from a decision point is traversed at least once.

$$\text{Branch Coverage} = \left(\frac{\text{Number of executed branches}}{\text{Total number of branches}} \right) \times 100$$

Example:

```
def check_eligibility(age):
    if age >= 18:
        print("Eligible to vote.")  # Statement A
    else:
        print("Not eligible.")      # Statement B
```

Scenario 1: Executing a single test case

- Test Case 1 Input: age = 20
- Execution Path: The condition age >= 18 is True. The code runs Statement A.
- Resulting Coverage:
  - Statement Coverage: 50% (1 out of 2 lines executed).
  - Branch Coverage: 50% (Only the True branch was tested; the False branch was missed).

Scenario 2: Achieving 100% Branch Coverage

To achieve full branch coverage, you must add a second test case to force the condition to evaluate to False.

- Test Case 2 Input: age = 15
- Execution Path: The condition evaluates to False, traversing the else block to execute Statement B.
- Resulting Coverage: 100% Branch Coverage (Both True and False outcomes have been tested).

Path Coverage: Ensures that all possible execution paths through the program are tested.To execute path coverage, testers analyze the code's internal structure and translate it into a visual model:

1. **Control Flow Graph (CFG):** The code is mapped into a graph where nodes represent code statements and edges represent the flow of control.
2. **Cyclomatic Complexity (V(G)):** This mathematical metric calculates the number of independent paths, determining the exact minimum number of test cases required to achieve base coverage. The formula generally used is: **V(G) = E - N + 2P** (Where E is the number of edges, N is the number of nodes, and P is the number of connected components).

Example Scenario

If you have a function with two independent if conditions, there are 4 possible distinct paths (2^2):

1. Condition 1 is True, Condition 2 is True
2. Condition 1 is True, Condition 2 is False
3. Condition 1 is False, Condition 2 is True
4. Condition 1 is False, Condition 2 is False

Path coverage mandates that a test case is designed and executed for all four combinations.

Condition Coverage: Ensures that each Boolean condition in a decision statement is evaluated as both true and false.

In complex software architectures, an expression like if (A && B) contains a single "decision" but contains two distinct "conditions" (A and B). Condition coverage isolates A and B, forcing the test suite to validate both binary possibilities for each variable independently.

$$\text{Condition Coverage} = \left(\frac{\text{Number of executed condition outcomes}}{\text{Total number of possible condition outcomes}} \right) \times 100$$

Example:

```
if (age >= 18 && hasTicket == true) {
   allowEntry();
}
```

Here, the code has two sub-conditions:

- Condition 1: age >= 18
- Condition 2: hasTicket == true [1]

To achieve 100% condition coverage, your test cases must toggle both conditions to True and False

Test Case	Input Values	Condition 1 (`age >= 18`)	Condition 2 (`hasTicket`)	Overall Decision Outcome
TC 1	`age = 20`, `hasTicket = true`	**True**	**True**	True (Enters `if`)
TC 2	`age = 15`, `hasTicket = false`	**False**	**False**	False (Skips `if`)

Analysis: With just two test cases, Condition 1 was evaluated as both True and False. Condition 2 was also evaluated as both True and False. Condition coverage is **100%**

Data Flow Testing is a white-box testing technique that focuses on the movement and usage of data within a program. It examines how variables are defined, utilized, and updated throughout execution to identify data-related defects.

This technique helps uncover problems such as uninitialized variables, variables that are never used, and improper data manipulation. By verifying the correct flow and handling of data, data flow testing improves the accuracy, reliability, and overall quality of the software.

Types of Data Flow Testing

- All-Defs Testing: Ensures every variable definition is linked to at least one use.
- All-Uses Testing: Ensures each variable definition is tested against all possible uses.
- All DU-Paths Testing: Ensures all definition-to-use paths are executed without redefinition.
- All P-Uses Testing: Ensures variables are tested in all decision-making conditions.
- All C-Uses Testing: Ensures variables are tested in all computational expressions and calculations.
- DU Pairs Testing: Ensures every valid definition-use pair of a variable is covered by test cases

Slicing Testing

Program Slicing is a software testing technique that selects only the relevant statements of a program that affect a particular variable or result. It helps testers focus on specific parts of the code for easier testing and debugging.

- It helps find and fix bugs quickly by analyzing only the relevant code.
- **Mark Weiser** first introduced program slicing as a static technique.
- Later, **Bogdan Korel** and **Janusz Laski** developed dynamic slicing, which works based on a specific program execution.

Types of Slicing

Static Slicing

Includes all statements that may affect a variable's value for any possible program execution.

- It usually produces a larger slice of the program.
- It considers all possible execution paths.

Dynamic Slicing

- **Dynamic Slicing:** Includes only the statements that actually affect a variable's value during a specific program execution.
- It usually produces a smaller slice of the program.
- It considers only one particular execution path.

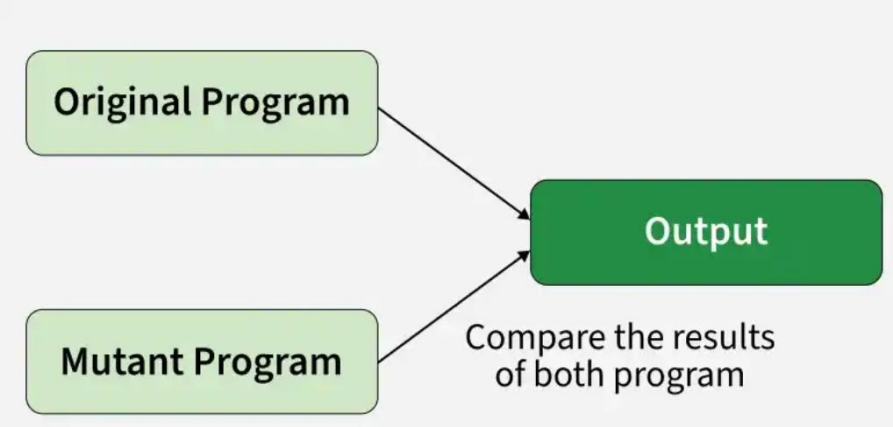
Backward Slice: A backward slice contains all statements that influence or affect the value of a selected variable at a specific point in the program.

Forward Slice: A forward slice contains all statements whose behavior or values are affected by a selected variable from a specific point onward in the program.

Mutation Testing

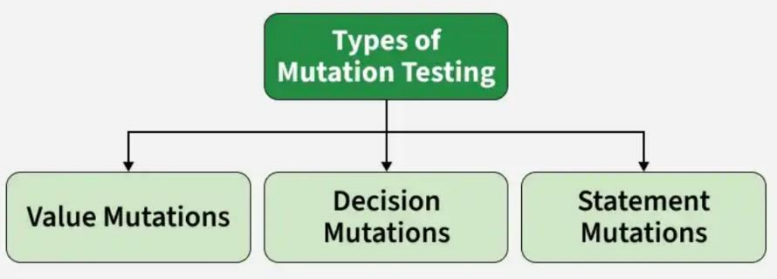
Mutation Testing is a white-box testing technique in which small changes (mutations) are made to the source code to evaluate the effectiveness of test cases. It helps identify weaknesses in test data and improves test quality.

- Small modifications are introduced into the code to create **mutants**.
- Existing test cases are executed to check whether they can detect the changes.
- It is mainly used during **unit testing** and helps uncover defects missed by other testing methods.
- If a test case detects a mutation, the mutant is called a **killed mutant**.
- If a mutation is not detected, the test cases need improvement.



Types of Mutation Testing

Mutation Testing can be classified into three main types based on the nature of the changes (mutations) introduced into the source code.



1. Value Mutations

In this type, the values of the constants are modified to identify issues in the code. A minor update is made on a very big value or a big value is updated to a smaller value.

Example:

```
int i = 100000089;
int j =  5678;
int k = 91011;
int c = (j * k) % i;
Updated Code:
int i = 1089;
int j =  5678;
int k = 91011;
int c = (j * k) % i;
```

2. Decision Mutations

In this type, the logical or arithmetic operators are updated to identify issues in the program source code.

Example :

```
if(i = j)
   k = 35;
else
   k = 50;
Updated Code:
if(i != j)
   k = 35;
else
   k = 50;
```

3. Statement Mutations

In this type, a statement is removed or replaced by another statement

```
if(i = j)
   k = 35;
else
   k = 50;
Updated Code:
if(i != j)
   m = 35;
else
   m = 50;
```

Tools used for Software Mutation Testing

The various tools used for the software mutation testing are listed below −

- PIT
- Insure
- Jester
- Jumble
- MuClipse

Structural Testing Process

The Structural Testing Process involves examining the internal structure of the software and designing test cases to ensure that different parts of the code are executed and verified**.**

- **Analyze the Source Code:** Examine the program structure, logic, and execution paths.
- **Design Test Cases:** Create test cases to cover statements, branches, and conditions.
- **Execute Test Cases:** Run the test cases and verify program behavior.
- **Measure Code Coverage:** Check how much of the code has been tested.
- **Identify and Fix Defects:** Locate errors and correct the uncovered issues.
- **Retest and Generate Reports:** Re-execute tests and generate coverage reports to confirm fixes.

Structural Testing Tools

- **JaCoCo:** Measures statement, branch, and line coverage in Java applications.
- **Cobertura:** Open-source tool for identifying code coverage and untested Java code.
- **GCov:** Coverage analysis tool for C/C++ programs using the GCC compiler.
- **BullseyeCoverage:** Commercial tool for measuring test coverage in C and C++ applications.
- **Istanbul (nyc):** Generates detailed code coverage reports for JavaScript applications.
- **Coverage.py:** Measures code coverage and execution in Python programs.

Advantages of Structural Testing

- **Ensures Code Coverage:** Verifies that different parts of the source code are tested.
- **Detects Defects Early:** Identifies coding and logical errors during development.
- **Improves Software Quality:** Increases software reliability and correctness.
- **Identifies Dead Code:** Finds unused or unreachable code segments.
- **Supports Test Optimization:** Helps create more effective test cases by revealing untested areas.
- **Facilitates Automation:** Can be automated using code coverage tools.
- **Enhances Code Reliability:** Verifies critical paths and conditions in the code.
- **Reduces Maintenance Costs:** Early defect detection lowers fixing costs and effort.