**Topic: White Box Testing Techniques**



**Hamza Mehmood (SP-21-110)**

**Jibran Naeem (SP-21-111)**

**Saad Rehman Raja (SP-21-118)**

*Submitted to*

**Dr. Sumaira Nazir**

*Submitted for the Presentation of subject of BS Software Engineering degree to the Faculty of Engineering & Computing*

**NATIONAL UNIVERSITY OF MODERN LANGUAGES**

**ISLAMABAD**

**December, 2023**

**TABLE OF CONTENT**

Contents

[1. Introduction 0](#_Toc153636281)

[2. Focus 0](#_Toc153636282)

[3. Why White Box testing 0](#_Toc153636283)

[3.1 Internal Code Examination 0](#_Toc153636284)

[3.2 Early Detection of Defects 0](#_Toc153636285)

[3.3 Code Coverage 0](#_Toc153636286)

[3.4 Security Assessment 1](#_Toc153636287)

[3.5 Optimizing Performance 1](#_Toc153636288)

[3.6 Integration Testing 1](#_Toc153636289)

[3.7 Quality Assurance 1](#_Toc153636290)

[3.8 Regulatory Compliance 1](#_Toc153636291)

[4. How is it different from Black Box testing 1](#_Toc153636292)

[4.1 White Box Testing 1](#_Toc153636293)

[4.1.1 Focus 1](#_Toc153636294)

[4.1.2 Access to Code 2](#_Toc153636295)

[4.1.3 Testing Levels 2](#_Toc153636296)

[4.1.4 Testing Techniques 2](#_Toc153636297)

[4.1.5 Testing Objectives 2](#_Toc153636298)

[4.2 Black Box Testing 2](#_Toc153636299)

[4.2.1 Focus 2](#_Toc153636300)

[4.2.2 Access to Code 2](#_Toc153636301)

[4.2.3 Testing Levels 2](#_Toc153636302)

[4.2.4 Testing Techniques 2](#_Toc153636303)

[4.2.5 Testing Objectives 3](#_Toc153636304)

[4.3 Comparison 3](#_Toc153636305)

[4.3.1 Knowledge 3](#_Toc153636306)

[4.3.2 Test Case Design 3](#_Toc153636307)

[4.3.3 Scope 3](#_Toc153636308)

[4.3.4 Applicability 3](#_Toc153636309)

[5. White Box Techniques 3](#_Toc153636310)

[5.1 Statement coverage: 3](#_Toc153636311)

[5.2 Branch Coverage: 4](#_Toc153636312)

[Description 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](#_Toc153636313)

[5.3 Condition Coverage: 4](#_Toc153636314)

[5.4 Multiple Condition Coverage: 4](#_Toc153636315)

[5.5 Basis Path Testing: 4](#_Toc153636316)

[6. Examples of White Box Testing Techniques 5](#_Toc153636317)

[6.1 Statement Coverage: 5](#_Toc153636318)

[6.2 Branch Coverage: 5](#_Toc153636319)

[6.3 Condition Coverage: 5](#_Toc153636320)

[6.4 Multiple Condition Coverage: 6](#_Toc153636321)

[6.5 Basis Path Testing: 6](#_Toc153636322)

[7. Benefits 7](#_Toc153636323)

[7.1 Sequencing Verification 7](#_Toc153636324)

[7.2 Identification of Flow Issues 7](#_Toc153636325)

[7.3 Enhanced Code Coverage 7](#_Toc153636326)

[7.4 Early Detection of Defects 7](#_Toc153636327)

[7.5 Improved Code Quality 7](#_Toc153636328)

[7.6 Effective Debugging 8](#_Toc153636329)

[7.7 Enhanced Confidence in Code 8](#_Toc153636330)

[8. Challenges 8](#_Toc153636331)

[8.1 Complexity of Code 8](#_Toc153636332)

[8.2 Code Changes and Maintenance 8](#_Toc153636333)

[8.3 Dependency Management 8](#_Toc153636334)

[8.4 Test Case Design 9](#_Toc153636335)

[8.5 Path Explosion 9](#_Toc153636336)

[8.6 Dynamic Environments 9](#_Toc153636337)

[8.7 Testing External Interactions 9](#_Toc153636338)

[8.8 Skill Requirements 9](#_Toc153636339)

[8.9 Resource Intensiveness 10](#_Toc153636340)

# Introduction

White box testing is a software testing methodology that focuses on examining the internal logic, structure, and code of a software application. Also known as clear box testing, transparent box testing, or glass box testing, this approach involves scrutinizing the software's internal components, including algorithms, data flow, and control flow, to ensure that they function as intended.

In white box testing, testers have access to the source code, design documents, and architecture details of the software being tested. This enables them to create test cases based on a deep understanding of the internal workings of the application. The primary goals of white box testing are to verify the correctness of the code, ensure that all paths of the code are executed, and identify any potential vulnerabilities or weaknesses in the software's design.

Key activities in white box testing include code analysis, unit testing, integration testing, and path testing. By thoroughly examining the internal structure of the software, white box testing helps uncover issues related to code quality, security, and overall software reliability. This method is crucial in ensuring that the software meets specified requirements, operates efficiently, and is robust enough to handle various scenarios.

# 2. Focus

Examines the internal structure and logic of your software, not just its external structure.

# 3. Why White Box testing

White box testing is employed for several important reasons in the software development and testing process. Here are some key reasons why white box testing is essential:

## 3.1 Internal Code Examination

- White box testing allows testers to examine the internal code, structure, and logic of the software. This deep dive into the internals helps identify issues that may not be apparent through other testing methods.

## 3.2 Early Detection of Defects

- By focusing on the code level, white box testing facilitates the early detection and resolution of defects. This helps in reducing the overall cost of fixing issues as they are discovered closer to their origin in the development cycle.

## 3.3 Code Coverage

- White box testing ensures comprehensive code coverage, meaning that all lines of code, branches, and paths are exercised during testing. This helps in confirming that the entire codebase is tested and potential issues are not overlooked.

## 3.4 Security Assessment

- Security vulnerabilities often reside in the code. White box testing is instrumental in identifying and addressing potential security issues, ensuring that the software is resilient against security threats.

## 3.5 Optimizing Performance

- White box testing allows for the evaluation of critical functions and algorithms, helping to optimize the performance of the software. This is crucial for ensuring that the application meets performance expectations.

## 3.6 Integration Testing

- White box testing is well-suited for integration testing, where the interactions between different components are examined. This ensures that the integrated system behaves as expected and that components work seamlessly together.

## 3.7 Quality Assurance

- By assessing the code quality, readability, and adherence to coding standards, white box testing contributes to overall software quality. This is essential for creating maintainable and sustainable software.

## 3.8 Regulatory Compliance

- In certain industries with strict regulatory requirements (e.g., finance, healthcare), white box testing helps ensure compliance by thoroughly validating the internal logic and data flow within the software.

# 4. How is it different from Black Box testing

# 

White box testing and black box testing are two distinct approaches to software testing, each with its own focus, objectives, and methodologies. Here are the key differences between white box testing and black box testing

## 4.1 White Box Testing

### 4.1.1 Focus

**Internal Structure** White box testing focuses on the internal logic, structure, and code of the software. Testers have knowledge of the internal workings of the application.

### 4.1.2 Access to Code

**Source Code Access** Testers have access to the source code, design documents, and architecture details, allowing for a detailed examination of the code base.

### 4.1.3 Testing Levels

**Unit, Integration, and System Testing** White box testing is conducted at multiple levels, including unit testing (testing individual components), integration testing (testing interactions between components), and system testing (testing the entire system).

### 4.1.4 Testing Techniques

**Code Coverage** White box testing emphasizes achieving high code coverage, ensuring that all lines of code, branches, and paths are executed during testing.

### 4.1.5 Testing Objectives

**Error Detection and Prevention** White box testing aims to detect and prevent errors in the code, optimizing the software for correctness, security, and performance.

## 4.2 Black Box Testing

### 4.2.1 Focus

**External Behavior** Black box testing focuses on the external behavior of the software without knowledge of its internal code. Testers treat the software as a "black box" with inputs and expected outputs.

### 4.2.2 Access to Code

**No Source Code Access** Testers do not have access to the source code. Testing is based on functional specifications, requirements, and user documentation.

### 4.2.3 Testing Levels

**System testing and Acceptance Testing** Black box testing is often performed at higher levels, such as system testing (testing the entire system) and acceptance testing (ensuring the software meets user requirements).

### 4.2.4 Testing Techniques

**Equivalence Partitioning, Boundary Value Analysis** Black box testing uses techniques like equivalence partitioning and boundary value analysis to design test cases based on the expected external behavior.

### 4.2.5 Testing Objectives

**Functional Validation** Black box testing is primarily concerned with validating that the software functions according to specified requirements. It assesses the software's correctness, completeness, and user acceptance.

## 4.3 Comparison

### 4.3.1 Knowledge

White box testing requires knowledge of the internal structure, while black box testing does not necessitate knowledge of the internal implementation.

### 4.3.2 Test Case Design

White box testing involves designing test cases based on internal logic, code flow, and structure. Black box testing designs test cases based on functional specifications and requirements.

### 4.3.3 Scope

White box testing provides detailed insights into the internal workings of the software. Black box testing assesses the software from a user's perspective without delving into internal details.

### 4.3.4 Applicability

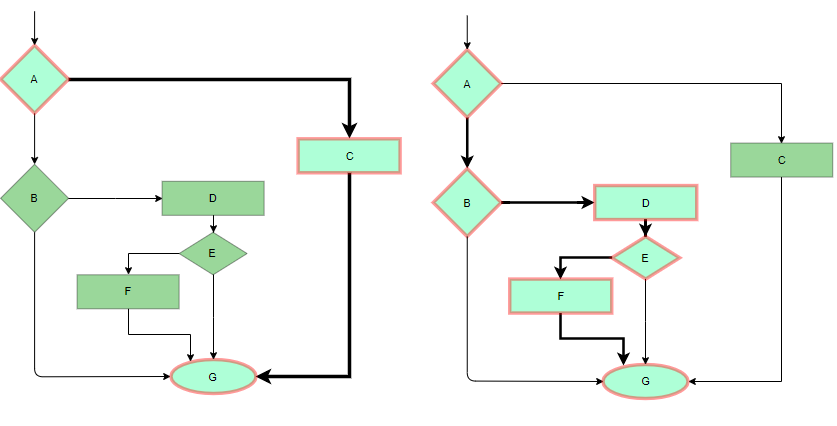
White box testing is suitable for early stages of development and for critical systems where internal logic is crucial. Black box testing is suitable for assessing the software's overall functionality and user satisfaction.

# 5. White Box Techniques

White box testing techniques involve the examination of the internal structure, logic, and code of a software application. These techniques aim to ensure that the internal components function correctly and efficiently. Here are some commonly used white box testing techniques:

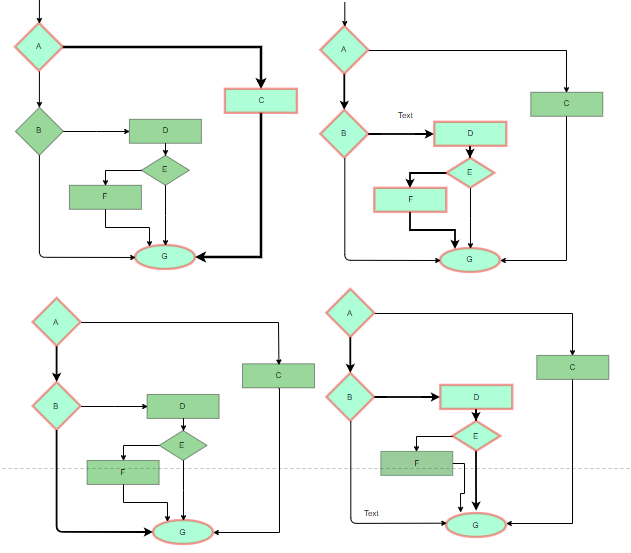
## 5.1 Statement coverage:

**Description** 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, it helps in pointing out faulty code.



## 5.2 Branch Coverage:

## Description 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.



## 5.3 Condition Coverage:

**Description** In this technique, all individual conditions must be covered as shown in the following **example:**

READ X, Y

IF(X == 0 || Y == 0)

PRINT ‘0’

#TC1 – X = 0, Y = 55

#TC2 – X = 5, Y = 0

## 5.4 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:**

READ X, Y

IF(X == 0 || Y == 0)

PRINT ‘0’

#TC1: X = 0, Y = 0

#TC2: X = 0, Y = 5

#TC3: X = 55, Y = 0

#TC4: X = 55, Y = 5

## 5.5 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:

Make the corresponding control flow graph

Calculate the cyclomatic complexity

Find the independent paths

Design test cases corresponding to each independent path

V(G) = P + 1, where P is the number of predicate nodes in the flow graph

V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes

V(G) = Number of non-overlapping regions in the graph

#P1: 1 – 2 – 4 – 7 – 8

#P2: 1 – 2 – 3 – 5 – 7 – 8

#P3: 1 – 2 – 3 – 6 – 7 – 8

#P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

# 6. Examples of White Box Testing Techniques

Certainly! Here are examples of some white box testing techniques:

## 6.1 Statement Coverage:

**Technique:** Ensure that each statement in the code is executed at least once during testing.

Example: If you have the following code:

**def add\_numbers(a, b):**

**result = a + b**

**print(result)**

**return result**

A test case might involve calling add\_numbers(2, 3) and ensuring that the print statement is executed.

## 6.2 Branch Coverage:

Technique: Aim to test each branch of the decision-making structure in the code.

Example: For the code snippet below:

**def is\_positive(number):**

**if number > 0:**

**return True**

**else:**

**return False**

Test cases should cover both the True and False branches, so testing with both positive and non-positive numbers is necessary.

## 6.3 Condition Coverage:

**def is\_triangle(a, b, c):**

**if a > 0 and b > 0 and c > 0:**

**if a + b > c and a + c > b and b + c > a:**

**return "It is a triangle."**

**else:**

**return "It is not a valid triangle."**

**else:**

**return "Invalid side lengths."**

**# Test cases for Condition Coverage**

**result1 = is\_triangle(3, 4, 5) # Valid triangle**

**result2 = is\_triangle(1, 1, 3) # Not a valid triangle**

## 6.4 Multiple Condition Coverage:

**def check\_number(x):**

**if x > 0 and x % 2 == 0:**

**return "Positive even number"**

**elif x > 0 and x % 2 != 0:**

**return "Positive odd number"**

**elif x < 0:**

**return "Negative number"**

**else:**

**return "Zero"**

**# Test cases for Multiple Condition Coverage**

**result1 = check\_number(6) # Positive even number**

**result2 = check\_number(7) # Positive odd number**

**result3 = check\_number(-2) # Negative number**

**result4 = check\_number(0) # Zero**

## 6.5 Basis Path Testing:

**def calculate\_discount(amount, is\_member):**

**if amount > 100 and is\_member:**

**discount = 0.1**

**elif amount > 50 or is\_member:**

**discount = 0.05**

**else:**

**discount = 0**

**total = amount - (amount \* discount)**

**return total**

**# Test cases for Basis Path Testing**

**result1 = calculate\_discount(120, True) # Path: 1 → 2 → 4**

**result2 = calculate\_discount(70, False) # Path: 1 → 3**

**result3 = calculate\_discount(40, True) # Path: 1 → 2 → 3**

**result4 = calculate\_discount(30, False) # Path: 1 → 3**

# 7. Benefits

Control flow testing offers several benefits in the software testing process. Here are some of the key advantages:

## 7.1 Sequencing Verification

**Benefit** Control flow testing ensures that statements in the code are executed in the correct order according to the defined control flow.

**Explanation** By systematically testing different paths through the code, control flow testing verifies that the program's logic follows the intended sequencing, helping to catch issues related to incorrect execution order.

## 7.2 Identification of Flow Issues

**Benefit** The technique helps uncover defects related to incorrect branching, looping behavior, or other control flow issues.

**Explanation** By designing test cases that cover various control flow paths, testers can identify and address issues related to unexpected behavior, ensuring that the software behaves as intended in different scenarios.

## 7.3 Enhanced Code Coverage

**Benefit** Control flow testing contributes to achieving higher coverage of control flow paths within the code.

**Explanation** The technique aims to test all possible paths through decision points, loops, and other control structures, leading to more comprehensive coverage. This, in turn, increases the likelihood of detecting defects and improving the overall quality of the code.

## 7.4 Early Detection of Defects

**Benefit** Control flow testing facilitates early detection of defects related to the sequencing and branching of code.

**Explanation** By identifying issues during the development and testing phases, control flow testing allows developers to address defects before they propagate into later stages of the software development lifecycle, reducing the cost and effort of fixing issues.

## 7.5 Improved Code Quality

**Benefit** The technique contributes to improved code quality by identifying and addressing control flow-related issues.

**Explanation** Ensuring that the control flow paths are correct and well-tested helps maintain a higher standard of code quality, reducing the likelihood of bugs and enhancing the overall reliability of the software.

## 7.6 Effective Debugging

**Benefit** Control flow testing aids in effective debugging by pinpointing the source of issues related to control flow.

**Explanation** When issues are detected, the systematic nature of control flow testing makes it easier for developers to trace and debug problems, leading to faster issue resolution.

## 7.7 Enhanced Confidence in Code

**Benefit** Successfully implementing control flow testing increases confidence in the correctness of the software's control flow.

**Explanation** By thoroughly testing different paths through the code, control flow testing provides assurance that the program's logic is sound and that it will perform as expected in various scenarios, contributing to increased confidence in the software's reliability.

# 8. Challenges

White-box testing techniques, while valuable for assessing the internal structure and logic of software, come with their own set of challenges. Here are some common challenges associated with white-box testing techniques:

## 8.1 Complexity of Code

**Challenge:** Testing complex code with intricate control flow, nested conditions, and large codebases can be challenging.

**Explanation:** Understanding and testing complex code structures may require significant time and effort. The more intricate the code, the more challenging it becomes to design comprehensive test cases that cover all possible paths.

## 8.2 Code Changes and Maintenance

**Challenge:** White-box tests may become outdated or irrelevant when code undergoes changes, leading to maintenance challenges.

**Explanation:\*** As the software evolves, maintaining existing white-box tests to align with code modifications can be time-consuming. Changes in the code may require corresponding updates to test cases, impacting the overall testing effort.

## 8.3 Dependency Management

**Challenge:** Handling dependencies between different components, modules, or units can be complex.

**Explanation:** White-box testing often requires knowledge of internal dependencies. Changes in one part of the code may have unintended consequences on dependent components, making it challenging to isolate and test individual units.

## 8.4 Test Case Design

**Challenge:** Designing effective test cases that cover all code paths and potential defects can be difficult.

**Explanation:** Generating test cases that achieve high code coverage and uncover potential issues requires a deep understanding of the code. Creating comprehensive test cases becomes especially challenging in scenarios with numerous decision points and complex control flow.

## 8.5 Path Explosion

**Challenge:** As the number of paths through the code increases, the number of test cases required for thorough testing can become impractical.

**Explanation:** Achieving 100% path coverage may be infeasible in real-world scenarios, leading to the challenge of deciding which paths are most critical to test while maintaining a reasonable testing effort.

## 8.6 Dynamic Environments

**Challenge:** Testing in dynamic environments with frequent changes and updates can be challenging.

**Explanation:** In agile development or continuous integration environments, code changes rapidly. Adapting white-box testing strategies to keep up with dynamic development cycles requires efficient coordination between development and testing teams.

## 8.7 Testing External Interactions

**Challenge:** White-box testing typically focuses on internal code behavior, making it less suitable for testing external interactions, such as database interactions, network communication, or external services.

**Explanation:** While white-box testing is excellent for assessing internal logic, it may not capture issues related to external dependencies, integration points, or environmental factors.

## 8.8 Skill Requirements

**Challenge:** Effective white-box testing often requires a high level of technical expertise.

**Explanation:** Testers need a deep understanding of programming languages, algorithms, and the underlying technology stack to design meaningful white-box tests. This can be a challenge in teams where testers have diverse backgrounds and skill levels.

## 8.9 Resource Intensiveness

**Challenge:** White-box testing, particularly techniques like mutation testing, can be computationally expensive and resource-intensive.

**Explanation:** Running numerous tests, especially when considering permutations of code changes in mutation testing, can consume significant computational resources, potentially slowing down the testing process.