Department of Computer Engineering

Academic Term: First Term 2023-24

Class: T.E /Computer Sem – V / Software Engineering

Practical No:	**9**
Title:	**Design test cases for performing white box testing**
Date of Performance:	5-10-23
Roll No:	9595
Team Members:	Atharva Dalvi

Rubrics for Evaluation:

Sr. No	**Performance Indicator**	**Excellent**	**Good**	**Below Average**	**Total Score**
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Theory Understanding(02)	02(Correct)	NA	01 (Tried)	
3	Content Quality (03)	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Questions (04)	04(done well)	3 (Partially Correct)	2(submitted)	

Department of Computer Engineering

Academic Term: First Term 2022-23

Class: T.E /Computer Sem – V / Software Engineering

Lab Experiment 09

Experiment Name:
Designing Test Cases for Performing White Box Testing in Software Engineering

Objective:
The objective of this lab experiment is to introduce students to the concept of White Box Testing, a testing technique that examines the internal code and structure of a software system.Students will gain practical experience in designing test cases for White Box Testing to verify the correctness of the software's logic and ensure code coverage.

Introduction:
White Box Testing, also known as Structural Testing or Code-Based Testing, involves assessing the internal workings of a software system. It aims to validate the correctness of the code,identify logic errors, and achieve maximum code coverage.

Lab Experiment Overview:
1. Introduction to White Box Testing: The lab session begins with an introduction to White Box Testing, explaining its purpose, advantages, and the techniques used, such as statement coverage,
branch coverage, and path coverage.
2. Defining the Sample Project: Students are provided with a sample software project along with its
source code and design documentation.
3. Identifying Test Scenarios: Students analyze the sample project and identify critical code segments, including functions, loops, and conditional statements. They determine the test scenarios
based on these code segments.
4. Statement Coverage: Students apply Statement Coverage to ensure that each statement in the code
is executed at least once. They design test cases to cover all the statements.
5. Branch Coverage: Students perform Branch Coverage to validate that every branch in the code,
including both true and false branches in conditional statements, is executed at least once. They design test cases to cover all branches.
6. Path Coverage: Students aim for Path Coverage by ensuring that all possible execution paths through the code are tested. They design test cases to cover different paths, including loop iterations and condition combinations.
7. Test Case Documentation: Students document the designed test cases, including the test scenario,
input values, expected outputs, and any assumptions made.
8. Test Execution: In a test environment, students execute the designed test cases and record the
results, analyzing the code coverage achieved.

9. Conclusion and Reflection: Students discuss the significance of White Box Testing in software quality assurance and reflect on their experience in designing test cases for White Box Testing.

Learning Outcomes:
By the end of this lab experiment, students are expected to:
Understand the concept and importance of White Box Testing in software testing.
Gain practical experience in designing test cases for White Box Testing to achieve code coverage.
Learn to apply techniques such as Statement Coverage, Branch Coverage, and Path Coverage in test case design.
Develop documentation skills for recording and organizing test cases effectively.
Appreciate the role of White Box Testing in validating code logic and identifying errors.

Pre-Lab Preparations:
Before the lab session, students should familiarize themselves with White
Box Testing concepts, Statement Coverage, Branch Coverage, and Path Coverage techniques.
Materials and Resources:
Project brief and details for the sample software project
Whiteboard or projector for explaining White Box Testing techniques
Test case templates for documentation

Conclusion: The lab experiment on designing test cases for White Box Testing equips students with essential skills in assessing the internal code of a software system. By applying various White Box Testing techniques, students ensure comprehensive code coverage and identify logic errors in the software. The experience in designing and executing test cases enhances their ability to validate code behavior and ensure code quality. The lab experiment encourages students to incorporate White Box Testing into their software testing strategies, promoting robust and high-quality software development. Emphasizing test case design in White Box Testing empowers students to contribute to software quality assurance and deliver reliable and efficient software solutions

In [1]:
```
"""Step 1: Defining the Sample Project

#We'll use a simple Python function for calculating the factorial of a number as our sample project"""

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

In [ ]:
```
""""Step 2: Identifying Test Scenarios

In this step, students identify critical code segments for the factorial function. For example, the critical code segments are
the conditional statement and the recursive call."""
```

In [2]:
```
"""Step 3: Statement Coverage

For statement coverage, you want to ensure that each statement is executed at least once"""

#Test cases for Statement Coverage
def test_factorial_statement_coverage():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(5) == 120
```

In [ ]:
```
"""Step 4: Branch Coverage

For branch coverage, ensure that every branch, including both true and false branches, is executed at least once."""

# Test cases for Branch Coverage
def test_factorial_branch_coverage():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(5) == 120
```

In [ ]:
```
"""Step 5: Path Coverage

Path coverage ensures that all possible execution paths through the code are tested. In this case, you would design test cases
for different path combinations"""

# Test cases for Path Coverage
def test_factorial_path_coverage():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(5) == 120
```

In [ ]:
```
"""Step 6: Test Case Documentation

Document the designed test cases, including the test scenario, input values, expected outputs, and any assumptions made."""
```

In [ ]:
```
# Test Case Documentation for the `factorial` function

# Test Scenario 1: Testing factorial(0)
# Input: n = 0
# Expected Output: 1
# Assumptions: The function will handle the base case (n = 0) correctly.
# Explanation: The factorial of 0 is defined as 1.

# Test Scenario 2: Testing factorial(1)
# Input: n = 1
# Expected Output: 1
# Assumptions: The function will correctly handle the case where n is 1.
# Explanation: The factorial of 1 is 1.

# Test Scenario 3: Testing factorial(5)
# Input: n = 5
# Expected Output: 120
# Assumptions: The function can correctly calculate the factorial of a positive integer.
# Explanation: The factorial of 5 is 5 * 4 * 3 * 2 * 1 = 120.
```

In [ ]:
```
"""Step 8: Test Execution

Execute the test cases in a test environment and record the results, analyzing the achieved code coverage."""
```

In [4]:
```python
import unittest

# Sample `factorial` function
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

class TestFactorial(unittest.TestCase):
    def test_factorial_0(self):
        self.assertEqual(factorial(0), 1)

    def test_factorial_1(self):
        self.assertEqual(factorial(1), 1)

    def test_factorial_5(self):
        self.assertEqual(factorial(5), 120)

# Create a test suite
test_suite = unittest.TestLoader().loadTestsFromTestCase(TestFactorial)

# Create a test runner
test_runner = unittest.TextTestRunner(verbosity=2)

# Run the tests and display the results
test_runner.run(test_suite)
```

```
test_factorial_0 (__main__.TestFactorial.test_factorial_0) ... ok
test_factorial_1 (__main__.TestFactorial.test_factorial_1) ... ok
test_factorial_5 (__main__.TestFactorial.test_factorial_5) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

Out[4]:
```
<unittest.runner.TextTestResult run=3 errors=0 failures=0>
```

POST:ABS

a) Generate white box test cases to achieve 100% statement coverage for a given code snippet.

Achieving 100% statement coverage in white-box testing means that every statement in the code is executed at least once. To generate test cases for 100% statement coverage, you need to consider all possible execution paths in the code. Here's a simple example of a code snippet and its associated white-box test cases to achieve 100% statement coverage:

Let's say we have the following code snippet in Python:

```
python
def is_prime(n):
    if n <= 1:
        return False
    elif n == 2:
        return True
    else:
        for i in range(2, n):
            if n % i == 0:
                return False
        return True
```

To achieve 100% statement coverage for this code, you need to create test cases that cover all the code paths, including the if-elif-else branches and the loop.

Here are test cases for 100% statement coverage:

1. Test Case 1: n <= 1
   - Input: n = 0
   - Expected Output: False

2. Test Case 2: n == 2
   - Input: n = 2
   - Expected Output: True

3. Test Case 3: n > 2 and prime
   - Input: n = 7
   - Expected Output: True

4. Test Case 4: n > 2 and not prime
   - Input: n = 9
   - Expected Output: False

5. Test Case 5: n > 2 and a large prime number
   - Input: n = 97
   - Expected Output: True

6. Test Case 6: n > 2 and a large non-prime number
   - Input: n = 100
   - Expected Output: False

These test cases ensure that every statement in the code is executed, covering all possible code paths. The test cases cover the conditions in the if-elif-else statements and the loop.

Please note that achieving 100% statement coverage is just one aspect of testing. In practice, you should also consider other types of testing, such as boundary testing, equivalence partitioning, and error-handling scenarios to ensure comprehensive code coverage and correctness.

b) Compare and contrast white box testing with black box testing, highlighting their respective strengths and weaknesses in different testing scenarios.

White-box testing and black-box testing are two distinct approaches to software testing, each with its own set of strengths and weaknesses. They are often used together to provide comprehensive test coverage. Here's a comparison of white-box testing and black-box testing, highlighting their respective attributes and when each is more suitable:

White-Box Testing:

1. *Focus*: White-box testing, also known as structural or glass-box testing, focuses on examining the internal structure and logic of the software. Testers need knowledge of the source code and its design.

2. *Testing Approach*: White-box testing is typically carried out by examining the code, its algorithms, and data structures to design test cases. It is code-centric.

3. *Strengths*:
   - Excellent for verifying code coverage, such as statement, branch, and path coverage.
   - Effective for finding logic errors, coding mistakes, and security vulnerabilities.
   - Ideal for unit testing and ensuring that individual components or functions work correctly.
   - It can uncover complex code interactions and edge cases.

4. *Weaknesses*:
   - May not detect requirements or design flaws, as it focuses on implementation details.
   - Requires access to the source code and specific technical knowledge.
   - Testing every possible code path can be time-consuming.
   - Tends to miss integration and system-level issues that black-box testing might catch.

Black-Box Testing:

1. *Focus*: Black-box testing examines the software's external behavior and functionality without knowledge of its internal workings. Testers are unaware of the code's structure.

2. *Testing Approach*: Black-box testers design test cases based on functional requirements, specifications, or user documentation. It is user-centric.

3. *Strengths*:
   - Effectively assesses whether the software meets its specified requirements.
   - Ideal for testing user interfaces, usability, and user experience.
   - Suitable for integration testing and system testing, as it evaluates the software as a whole.
   - Testers do not need to understand the code, making it accessible to non-developers.

4. *Weaknesses*:
   - May miss subtle defects in the code or logic that are not apparent from the specifications.
   - Limited ability to uncover security vulnerabilities or design issues.
   - Achieving complete code coverage is not guaranteed.
   - Test cases are based on requirements, and if requirements are incomplete or incorrect, important issues may be overlooked.

Scenario-Based Comparison:

- *Unit Testing*: White-box testing is commonly used for unit testing to ensure that individual components or functions work correctly.

- *System Testing*: Black-box testing is more suitable for system testing, as it evaluates the software as a whole, focusing on functional and non-functional requirements.

- *Security Testing*: White-box testing is essential for uncovering security vulnerabilities, as testers can inspect the code for potential weaknesses.

- *User Interface Testing*: Black-box testing is ideal for evaluating the user interface, usability, and overall user experience.

- *Integration Testing*: A combination of both white-box and black-box testing is often used for integration testing to ensure that components interact correctly and meet functional requirements.

In practice, a balanced approach combining both white-box and black-box testing is often the most effective way to ensure software quality and reliability. The choice between the two methods depends on the testing objectives, the stage of the software development lifecycle, and the resources available.

c) Analyze the impact of white box testing on software quality, identifying its potential to uncover complex logic errors and security vulnerabilities.

White-box testing plays a crucial role in improving software quality by delving into the internal structure and logic of the software. Its impact on software quality is significant, as it has the potential to uncover complex logic errors and security vulnerabilities. Here's an analysis of the impact of white-box testing in these areas:

1. *Uncovering Complex Logic Errors*:

- *Code Coverage*: White-box testing, through techniques like statement coverage, branch coverage, and path coverage, ensures that all lines of code are executed and all logical branches are tested. This comprehensive coverage helps uncover complex logical errors that may remain hidden during black-box testing.

- *Path Analysis*: White-box testing examines different execution paths through the code. This is particularly valuable for identifying complex conditional logic errors, infinite loops, or unexpected edge cases.

- *Boundary and Equivalence Testing*: White-box testing can target specific boundaries and equivalence classes, helping to expose issues related to boundary conditions and incorrect handling of input ranges.

- *Integration Testing*: White-box testing is essential for integration testing, where complex interactions between various components are examined. It helps identify issues arising from the integration of different modules or services.

2. *Detecting Security Vulnerabilities*:

- *Code Inspection*: White-box testing allows security experts to inspect the code and identify potential security vulnerabilities. They can analyze how data is handled, access control mechanisms, and authentication procedures.

- *Secure Coding Practices*: White-box testing encourages the use of secure coding practices. Testers can look for common coding mistakes that lead to vulnerabilities, such as SQL injection, cross-site scripting (XSS), or buffer overflows.

- *Access Control*: It can verify if access control mechanisms are correctly implemented and if there are any unauthorized paths to sensitive data or functionalities.

- *Static Analysis*: White-box testing often involves static code analysis tools that can identify security vulnerabilities, such as code injection, information leakage, or insecure dependencies.

3. *Quality Assurance*:

- White-box testing helps maintain code quality by identifying code smells, code duplications, and architectural issues. This can lead to better maintainability and easier troubleshooting in the long run.

- By ensuring the integrity of the code and the correctness of its logic, white-box testing contributes to software reliability, which is a key aspect of software quality.

4. *Early Issue Identification*:

   - White-box testing can be performed early in the development process, making it easier and less costly to address issues and defects before they become deeply ingrained in the codebase.

5. *Reduction of Post-release Issues*:

   - Uncovering and fixing complex logic errors and security vulnerabilities during white-box testing can significantly reduce the number of issues discovered post-release, leading to better customer satisfaction and lower maintenance costs.

In conclusion, white-box testing is an integral part of ensuring software quality. Its ability to uncover complex logic errors and security vulnerabilities is invaluable for preventing critical issues, improving software reliability, and providing a robust and secure software product. When combined with other testing approaches, white-box testing contributes to a comprehensive quality assurance strategy.