Department of Computer Engineering

Academic Term: First Term 2023-24

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

Practical No:	**5**
Title:	**Estimating project cost using COCOMO Model**
Date of Performance:	**05-09-2023**
Roll No:	**9639**
Team Members:	**Jenny Lopes, Janvi Naik**

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)	

Signature of the Teacher:

Lab Experiment 08

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

Objective: The objective of this lab experiment is to introduce students to the concept of Black Box Testing, a testing technique that focuses on the functional aspects of a software system without examining its internal code. Students will gain practical experience in designing test cases for Black Box Testing to ensure the software meets specified requirements and functions correctly.
Introduction: Black Box Testing is a critical software testing approach that verifies the functionality of a system from an external perspective, without knowledge of its internal structure. It is based on the software's specifications and requirements, making it an essential part of software quality assurance.

Lab Experiment Overview:

1. Introduction to Black Box Testing: The lab session begins with an introduction to Black Box Testing, explaining its purpose, advantages, and the types of tests performed, such as equivalence partitioning, boundary value analysis, decision table testing, and state transition testing.
2. Defining the Sample Project: Students are provided with a sample software project along with its functional requirements, use cases, and specifications.
3. Identifying Test Scenarios: Students analyze the sample project and identify test scenarios based on its requirements and use cases. They determine the input values, expected outputs, and test conditions for each scenario.
4. Equivalence Partitioning: Students apply Equivalence Partitioning to divide the input values into groups that are likely to produce similar results. They design test cases based on each equivalence class.
5. Boundary Value Analysis: Students perform Boundary Value Analysis to determine test cases that focus on the boundaries of input ranges. They identify test cases near the minimum and maximum values of each equivalence class.
6. Decision Table Testing: Students use Decision Table Testing to handle complex logical conditions

in the software's requirements. They construct decision tables and derive test cases from different combinations of conditions.

7. State Transition Testing: If applicable, students apply State Transition Testing to validate the software's behavior as it moves through various states. They design test cases to cover state transitions.

8. Test Case Documentation: Students document the designed test cases, including the test scenario,

input values, expected outputs, and any preconditions or postconditions.

9. Test Execution: In a simulated test environment, students execute the designed test cases and record the results.

10. Conclusion and Reflection: Students discuss the importance of Black Box Testing in software quality assurance and reflect on their experience in designing test cases for Black Box Testing.

Learning Outcomes: By the end of this lab experiment, students are expected to:

- Understand the concept and significance of Black Box Testing in software testing.
- Gain practical experience in designing test cases for Black Box Testing based on functional requirements.
- Learn to apply techniques such as Equivalence Partitioning, Boundary Value Analysis, Decision
- Table Testing, and State Transition Testing in test case design.
- Develop documentation skills for recording and organizing test cases effectively.
- Appreciate the role of Black Box Testing in identifying defects and ensuring software functionality.

Postlab: Design test cases for performing black box testing:

a) **Create a set of black box test cases based on a given set of functional requirements, ensuring adequate coverage of different scenarios and boundary conditions.**

   Black box testing is a method of testing where the tester focuses on the functionality of the software application without knowledge of its internal code. To create a set of black box test cases based on functional requirements, it's essential to cover different scenarios and boundary conditions. Below, I'll provide a hypothetical example of functional requirements for a simple calculator application and then create black box test cases for it.

   Functional Requirements:
   A simple calculator application should perform the following operations:
   1. Addition (+): Add two numbers and display the result.
   2. Subtraction (-): Subtract two numbers and display the result.
   3. Multiplication (*): Multiply two numbers and display the result.
   4. Division (/): Divide two numbers and display the result.

5. Clear (C): Clear the calculator's memory.
6. Memory Recall (MR): Display the value stored in memory.
7. Memory Store (MS): Store the current result in memory.

Black Box Test Cases:

1. Addition Functionality Testing:
   - Test Case 1: Add two positive integers.
   - Test Case 2: Add two negative integers.
   - Test Case 3: Add a positive integer and a negative integer.
   - Test Case 4: Add zero to a positive integer.
   - Test Case 5: Add zero to a negative integer.
   - Test Case 6: Add two floating-point numbers.
   - Test Case 7: Add a large positive integer to a large negative integer.

2. Subtraction Functionality Testing:
   - Test Case 8: Subtract a positive integer from another positive integer.
   - Test Case 9: Subtract a negative integer from a positive integer.
   - Test Case 10: Subtract a positive integer from a negative integer.
   - Test Case 11: Subtract zero from a positive integer.
   - Test Case 12: Subtract a positive integer from zero.
   - Test Case 13: Subtract two floating-point numbers.
   - Test Case 14: Subtract a large negative integer from a large positive integer.

3. Multiplication Functionality Testing:
   - Test Case 15: Multiply two positive integers.
   - Test Case 16: Multiply a negative integer by a positive integer.
   - Test Case 17: Multiply zero by a positive integer.
   - Test Case 18: Multiply two floating-point numbers.
   - Test Case 19: Multiply a large positive integer by a large negative integer.

4. Division Functionality Testing:
   - Test Case 20: Divide a positive integer by another positive integer.
   - Test Case 21: Divide a negative integer by a positive integer.
   - Test Case 22: Divide zero by a positive integer.
   - Test Case 23: Divide a positive integer by zero (expect error handling).
   - Test Case 24: Divide a floating-point number by another floating-point number.
   - Test Case 25: Divide a large positive integer by a large negative integer.

5. Memory Functionality Testing:
   - Test Case 26: Store a result in memory using the MS function.
   - Test Case 27: Recall the stored value from memory using MR.
   - Test Case 28: Clear the memory using the C function.

b) Evaluate the effectiveness of black box testing in uncovering defects and validating the software's functionality, comparing it with other testing techniques.

Black box testing is a valuable testing technique for uncovering defects and validating software functionality. However, its effectiveness depends on the context, objectives, and the specific needs of the software being tested. Here's an evaluation of the effectiveness of black box testing compared to other testing techniques:

Advantages of Black Box Testing:

1. Independence from Code: Black box testing focuses on the software's external behavior and functionality, which means testers do not need knowledge of the internal code. This makes it ideal for testing when the development team is separate from the testing team.

2. User Perspective: Black box testing simulates how users interact with the software. Testers approach the system as end-users would, helping to ensure that the software meets user expectations.

3. Functional and Behavioral Testing: It is effective for verifying that the software meets its functional requirements and behaves as intended. It helps identify issues related to incorrect calculations, broken links, missing features, or incorrect outputs.

4. Validation of Requirements: Black box testing ensures that the software fulfills its requirements, making it a useful technique for validating the software against its specifications.

5. Testing with Limited Information: Testers can create test cases based solely on the software's requirements, making it possible to test a system without detailed knowledge of its underlying code.

Limitations of Black Box Testing:

1. Limited Coverage of Internal Logic: Black box testing doesn't directly test the internal logic or code structure of the software. It may miss certain defects related to specific lines of code or code paths.

2. Inadequate for White Box Testing: If you need to verify the logic within the code, identify security vulnerabilities, or examine code coverage, black box testing is not the appropriate technique. White box testing or code reviews are better suited for such purposes.

3. May Not Find Integration Issues: While black box testing verifies individual components or functions, it may not effectively uncover issues that arise from the integration of multiple components or systems.

4. Limited in Discovering Non-functional Issues: It may not be effective in uncovering non-functional issues like performance bottlenecks, memory leaks, or security vulnerabilities. Specialized testing techniques like performance testing, security testing, or load testing are better suited for these concerns.

Comparison with Other Testing Techniques:

- White Box Testing: White box testing is effective for verifying the internal logic of the software and identifying issues at the code level. It complements black box testing by revealing defects that black box testing might miss.

- Gray Box Testing: Gray box testing combines elements of both black box and white box testing. Testers have partial knowledge of the internal code, making it a balanced approach for certain testing scenarios.

- Automated Testing: Automated testing, whether black box or white box, can be highly efficient for regression testing and executing repetitive test cases. Black box automated testing tools are particularly useful for functional testing.

- User Acceptance Testing (UAT): UAT is a specialized form of black box testing where end-users or stakeholders validate the software against their requirements. It's essential for ensuring the software aligns with user expectations.

c) Assess the challenges and limitations of black box testing in ensuring complete test coverage and discuss strategies to overcome them.

Black box testing has its share of challenges and limitations when it comes to ensuring complete test coverage. Test coverage refers to the extent to which a test suite examines the functionality of the software and the data inputs it can accept. Here are some challenges and strategies to overcome them:

Challenges and Limitations of Black Box Testing for Complete Test Coverage:

1. Inadequate Test Case Design: Test cases may not cover all possible scenarios, leading to gaps in coverage. Testers may overlook edge cases, boundary conditions, and rare scenarios.

2. Lack of Code Visibility: Black box testers have no visibility into the internal code, making it difficult to assess code paths and logic that may be critical to testing coverage.

3.Complex Combinations: When dealing with software that accepts various inputs and parameters, it can be challenging to design test cases to cover all possible combinations effectively.

4. Dynamic Systems: In dynamic systems where software behavior changes over time, static black box test cases may not adapt to new behaviors.

5. Undefined Requirements: If the software lacks clear and comprehensive requirements, it becomes difficult to design test cases that ensure complete coverage.

Strategies to Overcome Black Box Testing Challenges:

1. Requirements Traceability: Ensure that test cases are traced back to specific requirements. This helps in confirming that each requirement is covered and aids in identifying gaps.

2. Use Equivalence Partitioning: Divide the input space into equivalence classes and test representative values from each class. This is a systematic way to address different data inputs and their effects.

3. Boundary Value Analysis: Focus on testing at the boundaries of input domains. Often, defects are more likely to occur at the edges of acceptable input values.

4. Error Guessing: Encourage testers to use their domain knowledge and intuition to guess where defects might be hiding. This can help find defects that may not be apparent from the requirements.

5. Combinatorial Testing: Use combinatorial testing techniques to efficiently cover multiple combinations of inputs. Tools like pairwise testing can help ensure comprehensive coverage while minimizing the number of test cases.

Conclusion: The lab experiment on designing test cases for Black Box Testing provides students with essential skills in verifying software functionality from an external perspective. By applying various Black Box Testing techniques, students ensure comprehensive test coverage and identify potential defects in the software. The experience in designing and executing test cases enhances their ability to validate software behavior and fulfill functional requirements. The lab experiment encourages students to incorporate Black Box Testing into their software testing strategies, promoting robust and high-quality software development. Emphasizing test case design in Black Box Testing empowers students to contribute to software quality assurance and deliver reliable and customer- oriented software solutions.