Department of Computer Engineering

Academic Term: First Term 2023-24

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

Practical No:	**8**
Title:	**Designing Test Cases for Performing Black Box Testing**
Date of Performance:	**7/10/23**
Roll No:	**9590**
Team Members:	

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)	

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.

Pre-Lab Preparations: Before the lab session, students should familiarize themselves with Black Box Testing concepts, Equivalence Partitioning, Boundary Value Analysis, Decision Table Testing, and State Transition Testing techniques.

Materials and Resources:

- Project brief and details for the sample software project
- Whiteboard or projector for explaining Black Box Testing techniques Test case templates for documentation

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 customeroriented software solutions.

Module Name:	Arduino Uno
Reference Document:	IEEE Paper
Created by:	Mann Patel, John Rajan George, Shreedhar Bhat
Date of creation:	21/07/2022
Date of review:	21/06/2023

TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/ FAIL)
TC_001	Testing the Gas Leakage	No Fire and No Smoke	1. Proper temperature should be maintained 2. Regular Maintenance expected	1. Turn on the system 2. Take source near sensor 3. Wait until detection (Output)	No Fire Detected No Smoke Detected	No ALERT , LED stays Green	No change	No change	PASS
TC_002	Testing the Gas Leakage	Fire and No Smoke	1. Proper temperature should be maintained 2. Regular Maintenance expected	1. Turn on the system 2. Take source near sensor 3. Wait until detection (Output)	Fire Detected No Smoke Detected	A message "ALERT " is sent, Buzzer goes off, LED turns Red	Buzzer goes off, LED turns red, Alert message is sent	goes off, LED turns red, Alert message is	PASS
TC_003	Testing the Gas Leakage	No Fire and Smoke	1. Proper temperature should be maintained 2. Regular Maintenance expected	1. Turn on the system 2. Take source near sensor 3. Wait until detection (Output)	No Fire Detected Smoke Detected	A message "ALERT " is sent, Buzzer goes off, LED turns Red	Buzzer goes off, LED turns red, Alert message is sent	goes off, LED turns red, Alert message is	PASS
TC_004	Testing the Gas Leakage	Fire and Smoke	1. Proper temperature should be maintained 2. Regular Maintenance expected	1. Turn on the system 2. Take source near sensor 3. Wait until detection (Output)	Fire Detected Smoke Detected	A message "ALERT " is sent, Buzzer goes off, LED turns Red	Buzzer goes off, LED turns red, Alert message is sent	goes off, LED turns red, Alert message is	PASS

Postlab:

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. Creating a set of black-box test cases based on functional requirements involves designing test cases without any knowledge of the internal code or system architecture. To ensure adequate coverage of different scenarios and boundary conditions, you can follow a structured approach. Here's a step-by-step guide on how to create a comprehensive set of black-box test cases:

Step 1: Understand the Functional Requirements

Before you start creating test cases, make sure you have a clear understanding of the functional requirements. This is essential for effective testing.

Step 2: Identify Test Scenarios

Identify different test scenarios based on the functional requirements. Consider the various ways users will interact with the system. Common scenarios include normal use cases, edge cases, and error conditions.

Step 3: Define Test Inputs

For each test scenario, define the inputs that the system will receive. Ensure that these inputs cover a wide range of possibilities, including valid and invalid data. This is where you will consider boundary conditions, extreme values, and outliers. **Step 4: Determine Expected Outputs**

Specify the expected outputs or outcomes for each test scenario. This includes both the expected results for valid inputs and the expected error messages or behaviors for invalid inputs.

Step 5: Create Test Cases

Now, create individual test cases based on the scenarios, inputs, and expected outputs you've identified. Each test case should be self-contained and cover a unique aspect of the system's functionality. Here's a template for a test case:

- **Test Case ID**: A unique identifier for the test case.
- **Test Scenario**: A brief description of the scenario being tested.
- **Test Input**: The specific input data, including any user actions or system inputs.
- **Expected Output**: The expected results or system responses.
- **Test Execution Steps**: Detailed steps for executing the test case.

Step 6: Prioritize Test Cases

Not all test cases are equally important. Prioritize your test cases based on factors like critical functionality, the frequency of use, and potential impact on users.

Step 7: Consider Equivalence Classes

Group similar inputs into equivalence classes and create test cases that represent each class. This reduces redundancy and ensures that you test a representative sample of input possibilities.

Step 8: Include Negative Testing

Ensure that you include test cases that intentionally use invalid or unexpected inputs to validate the system's error-handling capabilities.

Step 9: Verify Traceability

Make sure that each test case can be traced back to a specific functional requirement. This helps ensure that all requirements are adequately tested.

Step 10: Review and Refine

Review your test cases with a team to identify any gaps or improvements. Refine the test cases as needed based on feedback.

Remember that the number of test cases you create will depend on the complexity of the system and the level of testing required. The goal is to strike a balance between thorough testing and efficiency. Automated testing tools can also be used to execute these test cases efficiently.

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 that plays a crucial role in uncovering defects and validating a software's functionality. However, its effectiveness, like any testing method, depends on the context and the specific goals of the testing process. Let's evaluate the effectiveness of black box testing in comparison to other testing techniques, such as white box testing and gray box testing:

1. Black Box Testing:

- **Pros:**
- **Independence from Implementation:** Black box testing focuses solely on the software's external behavior and functionality without any knowledge of its internal code or architecture. This independence allows for an unbiased evaluation of the software, making it ideal for functional testing and requirements validation. - **User-Centric:** Black box testing simulates the user's perspective, ensuring that the software meets its intended purpose and user expectations.
- **Covers a Wide Range of Scenarios:** Black box testing can cover various scenarios, including normal use cases, boundary conditions, error conditions, and usability testing.
- **Ease of Use:** It's often easier to design and execute black box test cases since you don't need access to the source code.

- **Cons:**
- **Limited Code Coverage:** Black box testing may not be effective in identifying certain structural defects or code-specific vulnerabilities, as it doesn't delve into the internal workings of the software.

2. White Box Testing:

- **Pros:**
- **In-Depth Code Inspection:** White box testing involves a detailed examination of the software's internal code and logic, making it highly effective at uncovering code-related defects, such as security vulnerabilities, logical errors, and dead code. - **Code Coverage Metrics:** White box testing can provide precise code coverage metrics, which help ensure that the entire codebase is exercised.
- **Early Detection of Code-Level Issues:** White box testing is particularly useful during the development phase for early defect detection.

-
Cons:
- **Dependence on Implementation Details:** White box testing requires knowledge of the internal code, which can lead to bias in the testing process. - **May Miss Functional Issues:** White box testing primarily focuses on code correctness, which may lead to missing functional or usability issues that are more apparent in black box testing.

3. Gray Box Testing:

Gray box testing is a hybrid approach that combines elements of both black box and white box testing. It involves having limited knowledge of the internal code or system architecture.

- **Pros:**
- **Balanced Approach:** Gray box testing strikes a balance between the benefits of both black box and white box testing. Testers have some knowledge of the code structure but not to the extent that it biases their testing.
- **Effective for Security Testing:** Gray box testing is particularly useful for security testing, where some knowledge of the code structure can help in identifying vulnerabilities.

- **Cons:**
- **May Not Achieve Deep Code Coverage:** Gray box testing might not achieve as deep code coverage as white box testing because it still maintains some level of abstraction from the code.

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

Black box testing, while a valuable testing method, has its challenges and limitations when it comes to ensuring complete test coverage. Here are some of the challenges and strategies to overcome them:

1. Incomplete Test Coverage:

Challenge: Black box testing focuses on testing the software's external behavior, and it may miss some internal code paths and logical conditions. It can be challenging to ensure complete coverage of all possible scenarios, especially in complex software.

Strategies to Overcome:

- **Use Requirement Traceability:** Link each test case to specific functional requirements. This helps ensure that all required functionalities are tested.

-
Equivalence Partitioning: Group similar inputs into equivalence classes and select test cases from each class to ensure a representative sample of input possibilities.
- **Boundary Value Analysis:** Pay special attention to boundary conditions, as these often lead to defects. Test values near the boundaries to increase the likelihood of finding issues.

2. Limited Error Discovery:

Challenge: Black box testing is generally effective in identifying what's wrong but may not provide insight into why an error occurred. It may reveal symptoms of defects but not their root causes.

Strategies to Overcome:
- **Error Logging and Reporting:** Even in black box testing, incorporate error logging and reporting mechanisms. These can help developers diagnose issues more effectively.
- **Collaboration with Developers:** When a defect is found, collaborate with developers to understand and address the root cause, especially when the cause might be related to the code's internal logic.

3. Difficulty in Testing Complex Algorithms:

Challenge: Black box testing might not be suitable for testing complex algorithms or mathematical functions, as it may not reveal the internal logic of these algorithms.

Strategies to Overcome:
- **Gray Box Testing:** If it's essential to test the internal logic of complex algorithms, consider a gray box testing approach where testers have some knowledge of the algorithm's logic.
- **Use of Mathematical Models:** In some cases, mathematical models can be created to predict the expected outputs for complex algorithms, which can then be compared with actual results in black box testing.

4. Lack of Code-Level and Security Testing:

Challenge: Black box testing primarily focuses on functional testing and may not be suitable for uncovering code-level vulnerabilities and security issues.

Strategies to Overcome:

-
- **Complementary Testing:** Combine black box testing with white box testing for a more comprehensive approach. White box testing is better suited for code-level and security testing.
Penetration Testing: For security assessments, consider penetration testing, which simulates real-world attacks and can reveal security vulnerabilities that are difficult to identify through black box testing alone.

5. Challenges in Identifying Dead Code:

Challenge: Black box testing may not effectively identify dead code, which is code that is no longer used but still exists in the software.

Strategies to Overcome:
- **Coverage Analysis Tools:** Use code coverage analysis tools in combination with black box testing to identify unexecuted code. These tools can help identify dead code segments.
- **Review of Code Changes:** Review code changes and updates regularly to identify and remove unnecessary or unused code.

6. Difficulty in Testing User Interface and Usability:

Challenge: Black box testing can struggle to comprehensively test the user interface and assess usability.

Strategies to Overcome:
- **Usability Testing:** Conduct separate usability testing with real users to assess the user interface and overall user experience.
- **Exploratory Testing:** Allow testers to explore the software freely, mimicking real user behavior and identifying usability issues.

In conclusion, black box testing is a valuable testing method, but it is not without its challenges, especially when aiming for complete test coverage. Combining it with other testing techniques and using specific strategies to address its limitations can help ensure a more thorough and effective testing process. The choice of testing techniques and strategies should be tailored to the specific goals and needs of the software being tested.