Department of Computer Engineering

Academic Term: First Term 2023-24

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

Practical No:	**8**
Title:	**Design test cases for performing black box testing**
Date of Performance:	**21-09-2023**
Roll No:	**9594**
Team Members:	**Janice D'Cruz, Slayde Sequeira, Aston Castelino, Chhand Chaughule**

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 customer oriented software solutions.

TEST CASE 1:

1	**Test Case ID**		SP_001	**Test Case Description**		Test the Login Functionality in smartbin app				
2	**Created By**		Janice	**Reviewed By**		Slayde		**Version**		1.0
3										
4	**QA Tester's Log**		Review from the app with version in 1.0							
5										
6	**Tester's Name**		Aston	**Date Tested**		September 26, 2023		**Test Case (Pass/Fail/Not Executed)**		Pass
7										
8	**S #**	**Prerequisites:**				**S #**	**Test Data**			
9	1	Access to app				1	Email id=crce.9592.ce@gmail.com			
10	2	stable internet connectivity				2	password=123456			
11	3					3				
12	4					4				
13										
14	**Test Scenario**	Verify on entering valid userid and password, the user can login								
15										
16	**Step #**	**Step Details**		**Expected Results**		**Actual Results**		**Pass / Fail / Not executed / Suspended**		
17										
18	1	Navigate to smartbin app		app should open		As Expected		Pass		
19	2	click on the login button		login model should appear		As Expected		Pass		
20	3	Enter Userid & Password		user should enter the credentials		As Expected		Pass		
21	4	Click Submit		user is logged in		As Expected		Pass		
22										
23										
24										
25										
26										
27										

Test Case 1 | Test Case 2 | Test Case 3

TEST CASE 2:

1	**Test Case ID**		SP_002	**Test Case Description**		Test the rewards Functionality in smartbin app				
2	**Created By**		Janice	**Reviewed By**		Chhand		**version**		1.0
3										
4	**QA Tester's Log**		review from the app with version in 1.0		version in 1.0					
5										
6	**Tester's Name**		Slayde	**Date Tested**		September 26, 2023		**Test Case (Pass/Fail/Not Executed)**		Pass
7										
8	**S #**	**Prerequisites:**				**S #**	**Test Data Requirement**			
9	1	Access to app				1	Email id=crce.9592.ce@gmail.com			
10	2	stable internet connectivity				2	password=123456			
11	3	user must be logged in to the website				3				
12	4					4				
13										
14	**Test Conditions**	verifying the search functionality by entering a product								
15										
16	**Step #**	**Step Details**		**Expected Results**		**Actual Results**		**Pass / Fail / Not executed / Suspended**		
17										
18	1	Navigate to smartbin app		app should open		As Expected		Pass		
19	2	Enter Userid & Password and sign in		user should enter the credentials		As Expected		Pass		
20	3	locate rewards		the rewards page should open		As Expected		Pass		
21	4	see if rewards are earned for dumping waste		the app will display all rewards given to users		As Expected		Pass		
22										
23										
24										
25										
26										
27										

Test Case 1 | Test Case 2 | Test Case 3

TEST CASE 3:

1	**Test Case ID**		SP_003	**Test Case Description**		Test the UID functionality in smartbin app				
2	**Created By**		Slayde	**Reviewed By**		Aston		**Version**		1.0
3										
4	**QA Tester's Log**		review from the app with version in 1.0							
5										
6	**Tester's Name**		Chhand	**Date Tested**		September 26, 2023		**Test Case (Pass/Fail/Not Executed)**		Pass
7										
8	**S #**	**Prerequisites:**				**S #**	**Test Data Requirement**			
9	1	Access to Chrome Browser				1	Email id=crce.9592.ce@gmail.com			
10	2	stable internet connectivity				2	password=123456			
11	3	user must be logged in to the website				3				
12	4					4				
13										
14	**Test Conditions**		verifying the search functionality by entering a product							
15										
16	**Step #**	**Step Details**		**Expected Results**		**Actual Results**		**Pass / Fail / Not executed / Suspended**		
17										
18	1	Navigate to smartbin app		app should open		As Expected		Pass		
19	2	Enter Userid & Password and sign in		user should enter the credentials		As Expected		Pass		
20	3	dump waste in bin and enter uid on app		user should throw waste in bin and enter 6 digit uid on app		As Expected		Pass		
21	4	uid is accepted		UID is accepted and rewards will be given to user		As Expected		Pass		
22										
23										

Test Case 1 | Test Case 2 | Test Case 3

Equivalence Partitioning Test Cases:

- Valid 6-digit UID
  - Input: 123456
  - Expected Output: User ID is accepted, and the user is rewarded within the app.
- Invalid UID with less than 6 digits
  - Input: 12345
  - Expected Output: User ID is rejected with an error message, and no rewards are given.
- Invalid UID with more than 6 digits
  - Input: 1234567
  - Expected Output: User ID is rejected with an error message, and no rewards are given.
- Empty UID field
  - Input: (Leave UID field empty)
  - Expected Output: User is prompted to enter a 6-digit UID, and no rewards are given.

Test Cases for Equivalence Partitioning:

Test Case ID	Test Case Name	Input	Expected Output
EP-1	Minimum valid UID	100000	User ID is accepted, and the user is rewarded within the app.
EP-2	Maximum valid UID	999999	User ID is accepted, and the user is rewarded within the app.
EP-3	UID just below the lower boundary	99999	User ID is rejected with an error message, and no rewards are given.
EP-4	UID just above the upper boundary	1000000	User ID is rejected with an error message, and no rewards are given.

POSTLABS:

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 involves testing a system without knowing its internal structure. Below is an example of functional requirements for an imaginary "Calculator".

Functional Requirements:

1. The calculator should perform basic arithmetic operations, including addition, subtraction, multiplication, and division.
2. It should support positive and negative numbers.
3. The calculator should be able to handle decimal numbers.
4. It must allow the user to clear the current input.
5. It should provide a result display to show the calculation outcome.
6. The calculator should handle boundary conditions, such as division by zero.
7. It should allow continuous calculations, where the result of one operation can be used as an input for the next operation.

Black Box Test Cases:

1. Addition:
   - Test Case 1: Verify that the calculator correctly adds two positive integers.
   - Test Case 2: Verify that the calculator correctly adds a positive integer and a negative integer.
   - Test Case 3: Verify that the calculator correctly adds two negative integers.
   - Test Case 4: Verify that the calculator correctly adds two decimal numbers.

2. Subtraction:
   - Test Case 5: Verify that the calculator correctly subtracts two positive integers.
   - Test Case 6: Verify that the calculator correctly subtracts a positive integer from a negative integer.
   - Test Case 7: Verify that the calculator correctly subtracts two negative integers.
   - Test Case 8: Verify that the calculator correctly subtracts two decimal numbers.

3. Multiplication:
   - Test Case 9: Verify that the calculator correctly multiplies two positive integers.
   - Test Case 10: Verify that the calculator correctly multiplies a positive integer by a negative integer.
   - Test Case 11: Verify that the calculator correctly multiplies two negative integers.
   - Test Case 12: Verify that the calculator correctly multiplies two decimal numbers.

4. Division:
   - Test Case 13: Verify that the calculator correctly divides a positive integer by another positive integer.
   - Test Case 14: Verify that the calculator correctly divides a positive integer by a negative integer.
   - Test Case 15: Verify that the calculator correctly divides a negative integer by a positive integer.
   - Test Case 16: Verify that the calculator correctly divides by zero (boundary condition).

5. Clear Function:
- Test Case 17: Verify that the clear function resets the calculator's input and result display.

6. Continuous Calculations:
- Test Case 18: Verify that the calculator allows continuous calculations by using the result of one operation as input for the next.

7. Boundary Conditions:
- Test Case 19: Verify that the calculator can handle very large input values for addition and multiplication.
- Test Case 20: Verify that the calculator can handle very small input values for subtraction and division.

8. Decimal Numbers:
- Test Case 21: Verify that the calculator correctly performs operations with decimal numbers.

These test cases cover a wide range of scenarios and boundary conditions for the calculator application, ensuring adequate coverage of its functionality.

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 and effective software testing technique for uncovering defects and validating software functionality. However, its effectiveness can vary depending on the context and the specific requirements of the testing process. Let's evaluate the effectiveness of black box testing and compare it with other testing techniques.

Effectiveness of Black Box Testing:

1. Uncovering Defects:
- Black box testing is effective at identifying defects related to incorrect functionality, boundary conditions, and input/output issues. Testers focus on what the software should do based on its specifications and requirements, and this can help find issues related to incorrect calculations, missing features, or erroneous outputs.
- It is particularly useful in uncovering integration issues, where different parts of the software may not interact as expected.

2. Validation of Functionality:
- Black box testing is well-suited for validating software functionality based on external requirements and user expectations. Testers verify that the software meets its intended purpose without needing knowledge of the internal code.
- It helps ensure that the software complies with the specified inputs, outputs, and behaviors outlined in the requirements.

3. Independence from Implementation Details:
- Testers do not need knowledge of the internal code, which is a significant advantage. This means black box testing can be carried out by individuals who are not developers and are not exposed to potential biases.

4. User-Centric Approach:

- Black box testing aligns well with the end-users' perspective, as it focuses on what the user experiences, making it a user-centric testing method.

5. Systematic Test Case Design:
- Various techniques, such as equivalence partitioning, boundary value analysis, and decision tables, can be used to design test cases systematically, which helps ensure good test coverage.

Comparison with Other Testing Techniques:

1. White Box Testing:
- White box testing examines the internal structure and logic of the software. It can be very effective in identifying code-level defects and uncovering issues related to code paths, but it might not be as effective in evaluating the software's behavior from a user's perspective.
- White box and black box testing are often complementary, and a combination of both can provide comprehensive coverage.

2. Gray Box Testing:
- Gray box testing combines elements of both black box and white box testing. Testers have partial knowledge of the internal code, allowing them to design more targeted test cases. It is effective when some understanding of the internal workings is required.

3. Regression Testing:
- Regression testing checks whether new changes introduce defects or break existing functionality. Black box testing can be very effective for regression testing because it focuses on the software's external behavior, ensuring that the intended functionality remains intact.

In summary, black box testing is highly effective in uncovering defects and validating software functionality, particularly from a user's perspective and when the internal code is not accessible. It's a valuable part of a comprehensive testing strategy, and its effectiveness can be enhanced when combined with other testing techniques, depending on the specific needs of the software project.

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 valuable, does have its challenges and limitations when it comes to ensuring complete test coverage. Test coverage refers to the extent to which a testing process has exercised a software system, and achieving complete test coverage can be challenging in black box testing. Here are the challenges and strategies to overcome them:

Challenges:
1. Limited Knowledge of the Internal Code:
- Black box testers don't have access to the internal code, which can make it challenging to design tests that cover every code path and edge case.

2. Inadequate Test Scenarios:
- It can be difficult to identify all possible test scenarios and boundary conditions without knowledge of the code structure.

3. Overlooking Integration Issues:
- Black box testing may not effectively address integration problems, where various components interact with each other. These issues are often missed as they require insight into the internal workings.

4. Inefficiency in Exploratory Testing:
   - Black box testing is less efficient for exploratory testing where testers need to experiment and explore the application extensively.

Strategies to Overcome Black Box Testing Limitations:

1. Collaboration with Developers:
   - Foster collaboration between testers and developers. While black box testers may not have access to the source code, they can still communicate with developers to gain insights into the software's architecture and potential areas of concern.
2. Requirements Analysis:
   - Thoroughly analyze the software requirements and design test cases based on these requirements. Review and clarify requirements to ensure a comprehensive understanding.
3. Boundary Value Analysis and Equivalence Partitioning:
   - Use boundary value analysis and equivalence partitioning to systematically design test cases that cover a range of inputs, including boundary conditions and error scenarios.
4. Risk-Based Testing:
   - Prioritize testing based on risk assessment. Focus more on critical and high-impact areas of the application. This ensures that critical parts of the software are thoroughly tested even when complete coverage is challenging.
5. Use of Testing Tools:
   - Leverage automated testing tools that can simulate different inputs and user interactions to perform a large number of test cases efficiently.
6. Pair Testing:
   - Employ pair testing, where a tester works closely with a developer. The tester can describe test scenarios to the developer, who can provide insights into potential issues and code paths.
7. Comprehensive Test Plan:
   - Create a well-documented test plan that includes clear objectives, test scenarios, and exit criteria. Regularly review and update the plan as new information becomes available.
8. Exploratory Testing as Supplement:
   - While black box testing might not be the most efficient method for exploratory testing, supplement it with dedicated exploratory testing sessions to uncover unexpected issues.
9. Continuous Feedback and Learning:
   - Foster a culture of continuous learning and improvement. Document lessons learned from testing, share them with the team, and apply those lessons to future testing efforts.
10. External Audit and Third-party Review:
   - Consider having an external audit or third-party review of the software to gain an independent perspective and uncover issues that internal testers might overlook.

In conclusion, while black box testing has limitations in achieving complete test coverage, a combination of strategies, including collaboration, thorough analysis, risk prioritization, and the use of appropriate tools, can help mitigate these challenges and ensure effective testing of the software. The key is to strike a balance between structured, requirement-based testing and exploratory approaches while making use of all available resources and knowledge.