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:	14 - 09 - 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)	

Signature of the Teacher:

Department of Computer

Engineering Academic Term: First

Term 2022-23

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

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.

Code:

```
import java.util.ArrayList;
import java.util.List;

class Donation {
   String name;
   double amount;
   double itemValue;

Donation(String name, double amount, double itemValue) {
   this.name = name;
   this.amount = amount;
   this.itemValue = itemValue;
}
}

class Donor {
   String name;
   String email;
   String phone;

Donor(String name, String email, String phone) {
   this.name = name;
   this.email = email;
   this.phone = phone;
}
}

class DonationSystem {
   List<Donor> donors;
   List<Donation> donations;

DonationSystem() {
donors = new ArrayList<>();
donations = new ArrayList<>();
}

void addDonor(String name, String email, String phone) {
   Donor donor = new Donor(name, email, phone);
   donors.add(donor);
System.out.println("New donor added: " + name);
}

void makeDonation(String donorName, double amount, double itemValue) {
   for (Donor donor : donors) {
if (donor.name.equals(donorName)) {
if (amount >= 2000 && amount <= 100000 && itemValue >= 1000 && itemValue <= 50000) {
Donation donation = new Donation(donorName, amount, itemValue); donations.add(donation);
```

```
System.out.println("Thank you, " + donorName + "! Your donation of $" + amount + ranges "
" and item valued at $" + itemValue + " has been received.");
} else {
System.out.println("Invalid donation amount or item value");
}
return;
}
}
System.out.println("Donor not found: " + donorName);
}

void listDonations() { System.out.println("Donations received:"); for (Donation donation : donations) {
System.out.println("Donor: " + donation.name + " - Amount: $" + donation.amount + " - Item Value: $"
+ donation.itemValue);
}
}

void listDonors() { System.out.println("List of Donors:"); for (Donor donor : donors) {
System.out.println("Name: " + donor.name + " - Email: " + donor.email + " - Phone: " + donor.phone);
}
}

public static void main(String[] args) {
DonationSystem donationSystem = new DonationSystem();

donationSystem.addDonor("John", "john@email.com", "123-456-7890");
donationSystem.addDonor("Alice", "alice@email.com", "987-654-3210");

donationSystem.makeDonation("John", 5000, 3000);
donationSystem.makeDonation("Alice", 150000, 8000); // Invalid amount
donationSystem.makeDonation("Bob", 50000, 200);           // Donor not found

donationSystem.listDonations(); donationSystem.listDonors();
}
}
```

1. Equivalence Partitioning:

It is the black-box technique that divides the input domain into classes of data from which test cases can be derived. Equivalence partitioning defines test cases that uncover classes of errors thereby reducing the no. of test cases that must be developed. If the input condition specifies a range, one valid and two invalid equivalence classes are defined. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined. If an input condition is boolean, one valid and one invalid equivalence class is defined. By applying these guidelines, test cases for each input domain can be developed.

E.g. A program that selects donation amount as 2000 to 100,000 so it will 3 partitions as amount

i. less than 2000
ii. between 2000 and 100,000
iii. greater than 100,000

Invalid	Valid	Invalid
1999 **Partition 1**	2000 100000 **Partition 2**	100001 **Partition 3**

And the program will accept items only in the range of 1000 to 50000. It will create 3 partitions as:

i. less than 1000
ii. between 1000 and 50000
iii. greater than 50000

Invalid	Valid	Invalid
999 **Partition 1**	1000 50000 **Partition 2**	50001 **Partition 3**

Test Cases for Equivalence Partitioning:

Test Case ID	Test Case Name	Input	Expected Output
EP-1	Valid Donation	Donor: "John", Amount: ₹5000, Item Value: ₹3000	"Thank you, John! Your donation of ₹5000 and item valued at ₹3000 has been received."
EP-2	Invalid Amount	Donor: "Alice", Amount: ₹150000, Item Value: ₹8000	"Invalid donation amount or item value. Please check the ranges."
EP-3	Invalid Item Value	Donor: "Bob", Amount: ₹50000, Item Value: ₹200	"Invalid donation amount or item value. Please check the ranges."
EP-4	Donor Not Found	Donor: "Eve", Amount: ₹5000, Item Value: ₹3000	"Donor not found: Eve"

2. Boundary Value Analysis:

Boundary Value Analysis (BVA) in black box testing is a technique that tests values on the edges of valid input ranges. It aims to identify potential errors at boundaries as they are more likely to trigger defects.

BVA complements Equivalence Partitioning and helps ensure thorough testing by focusing on critical boundary values for input parameters or conditions. BVA assesses values just inside and outside the valid range to uncover potential software issues, making it an efficient and effective testing method for black box testing.

In our online donation system program it checks the boundary conditions for donating the amount which ranges between 2000-10,0000

Boundary Value analysis for money to be donated:
For Range 2000 to 10,0000

Invalid (Min -1)	**Valid (Min,+Min,-Max,Max)**	**Invalid (Max+1)**
1999	2000, 2001, 99999, 100000	100001

Boundary Value analysis for items to be donated:
For Range 1000 to 50,000

Invalid (Min -1)	**Valid (Min,+Min,-Max,Max)**	**Invalid (Max+1)**
999	1000,1001,49999,50000	50001

Test Cases for Boundary Value Analysis:

Test Case ID	Test Case Name	Input	Expected Output
BVA-1	Minimum Valid Amount and Item Value	Donor: "John", Amount: ₹2000, Item Value: ₹1000	"Thank you, John! Your donation of ₹2000 and item valued at ₹1000 has been received."
BVA-2	Just Below Minimum Valid Amount	Donor: "Alice", Amount: ₹1999, Item Value: ₹3000	"Invalid donation amount or item value. Please check the ranges."
BVA-3	Just Above Maximum Valid Amount	Donor: "Bob", Amount: ₹100001, Item Value: ₹2000	"Invalid donation amount or item value. Please check the ranges."
BVA-4	Minimum Valid Item Value	Donor: "Eve", Amount: ₹5000, Item Value: ₹1000	"Thank you, Eve! Your donation of ₹5000 and item valued at ₹1000 has been received."
BVA-5	Just Below Minimum Valid Item Value	Donor: "Grace", Amount: ₹3000, Item Value: ₹999	"Invalid donation amount or item value. Please check the ranges."
BVA-6	Just Above Maximum Valid Item Value	Donor: "Sam", Amount: ₹6000, Item Value: ₹50001	"Invalid donation amount or item value. Please check the ranges."
BVA-7	Donor Not Found	Donor: "Mike", Amount: ₹5000, Item Value: ₹3000	"Donor not found: Mike"

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.

Understand the Requirements: Begin by thoroughly understanding the functional requirements of the software. This involves reviewing the software specification documents and any other relevant information.

Identify Test Scenarios: Identify different scenarios based on the requirements. These scenarios should cover a range of inputs, conditions, and expected outputs. Consider positive and negative scenarios.

Boundary Conditions: Pay special attention to boundary conditions and edge cases. These are critical for ensuring comprehensive testing.

Equivalence Partitioning: Group input data into equivalence classes, which represent sets of inputs that should produce the same behavior. Test at least one case from each equivalence class.

Decision Tables: Create decision tables to cover all possible combinations of conditions and actions, especially in cases of complex conditional logic.

Use Case Testing: If applicable, create test cases based on use cases that represent real-world interactions with the software.

Error Handling: Include test cases that focus on error-handling scenarios, such as input validation and handling unexpected situations.

State Transitions: For systems with states, create test cases that cover state transitions and ensure the system behaves correctly at each state.

Interface Testing: Test the interfaces where the software interacts with external systems or components. Ensure data is correctly passed and received.

Performance and Stress Testing: Include test cases to evaluate the software's performance under load, stress, or other conditions that may affect 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 effective in many ways, but it's essential to compare it to other testing techniques to understand its strengths and weaknesses:

Defect Uncovering: Black box testing is effective in uncovering defects related to incorrect functionality, usability, security, and compatibility issues.

User Perspective: It focuses on testing the software from a user's perspective, ensuring it meets user expectations.

Independence: Testers do not need to know the internal code, making it independent of the programming language or implementation.

Validation: It validates that the software functions as specified in the requirements.

However, black box testing has limitations:

Limited Code Coverage: It may not fully exercise all paths within the code, potentially missing certain defects.

Limited Control: Testers have limited control over the internal logic and data, making it challenging to target specific code segments.

Integration Challenges: It may struggle to uncover integration issues between different components.

Comparing it with white box testing (which focuses on internal code structure) and gray box testing (combining aspects of both), black box testing is less effective at code-level defects but highly effective for functional and user experience testing.

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

Challenges in black box testing include:

Incomplete Coverage: Ensuring comprehensive test coverage can be challenging. To overcome this, prioritize critical and high-risk areas and use techniques like pairwise testing to reduce the number of test cases needed.

Lack of Internal Knowledge: Testers may not have access to the code or detailed knowledge of the software's internal structure. Encourage collaboration and knowledge sharing between development and testing teams to bridge this gap.

Data Variability: Handling a wide range of input data can be complex. Use data generation tools or data-driven testing to address this challenge.

Dynamic Behavior: For software with dynamic behavior, it's challenging to anticipate all possible states. Use state transition testing and exploratory testing to address dynamic scenarios.

Regression Testing: Maintain a strong regression test suite to ensure that changes in the software don't introduce new defects.