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:	**25-09-2023**
Roll No:	**9617**
Team Members:	**Joshua Lewis**

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 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.

CODE: The following code is use for the swipe feature in our app.

```
package com.example.ngofinder.Adapter;

import android.content.Context;
import android.widget.Toast;

import androidx.annotation.NonNull;

import com.example.ngofinder.Model.EventModel;
import com.example.ngofinder.Model.RegisteredUserModel;
import com.example.ngofinder.User;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;
import com.wenchao.cardstack.CardStack;

public class yourListener implements CardStack.CardEventListener {
    private String ngoId;
    private String eventId;
    String name, status;
    FirebaseDatabase database;
    FirebaseAuth auth;
    private Context context;
    private MyCardStackAdapter mAdapter;

    public yourListener(String ngoId, String eventId,Context context,MyCardStackAdapter
adapter) {
        this.ngoId = ngoId;
```

```
            this.eventId = eventId;
            this.context = context;
            this.mAdapter = adapter;
        }
        @Override
        public boolean swipeEnd(int direction, float distance) {
            if(direction==1 || direction==3)
            {
                registerforEvent(ngoId, eventId);
            }else {
                Toast.makeText(context, "Ah! Next Time Surely?!",
Toast.LENGTH_SHORT).show();
            }

            return (distance > 300) ? true : false;
        }

        @Override
        public boolean swipeStart(int direction, float distance) {
            return true;
        }

        @Override
        public boolean swipeContinue(int direction, float distanceX, float distanceY) {

            return true;
        }

        @Override
        public void discarded(int id, int direction) {
            int currentPosition = mAdapter.getCount() - 1; // Get the current card position
            mAdapter.mSwipedPositions.add(currentPosition); // Add the position to the set of
swiped positions
            mAdapter.mEventList.remove(currentPosition); // Remove the swiped card from the
list
            mAdapter.notifyDataSetChanged(); // Notify the adapter that the data set has
changed
        }

        @Override
        public void topCardTapped() {
            // This callback invoked when a top card is tapped by user.
        }

        private void registerforEvent(String ngoId, String eventId) {
            // Get database reference
            FirebaseDatabase database = FirebaseDatabase.getInstance();
            DatabaseReference eventRef =
database.getReference().child("NGOS").child(ngoId).child("Events").child(eventId);

            eventRef.addListenerForSingleValueEvent(new ValueEventListener() {
```

```
                @Override
                public void onDataChange(@NonNull DataSnapshot snapshot) {
                    EventModel event = snapshot.getValue(EventModel.class);
                    long count = snapshot.child("Registered Users").getChildrenCount();
                    int limit = event.getLimit();

                    if (count < limit) {
                        // Get user details
                        String userId =
FirebaseAuth.getInstance().getCurrentUser().getUid();

database.getReference().child("Users").child(userId).addListenerForSingleValueEvent(new
ValueEventListener() {
                            @Override
                            public void onDataChange(@NonNull DataSnapshot snapshot)
{
                                User user = snapshot.getValue(User.class);
                                name = user.getName();
                                status = "Approved";

                                // Create registered user object and add to database
                                RegisteredUserModel reg = new
RegisteredUserModel(name, status);
                                reg.setTimeStamp(System.currentTimeMillis());

database.getReference().child("NGOS").child(ngoId).child("Events").child(eventId).child("Re
gistered Users").child(userId).setValue(reg);

                                // Update user's registered events

database.getReference().child("Users").child(userId).child("EventsRegistered").child(ngoId).c
hild(eventId).setValue(event);

                                Toast.makeText(context, "You have successfully
registered", Toast.LENGTH_SHORT).show();
                            }

                            @Override
                            public void onCancelled(@NonNull DatabaseError error) {
                                // Handle any errors
                            }
                        });
                    } else {
                        Toast.makeText(context, "Limit reached! Unsuccessful",
Toast.LENGTH_SHORT).show();
                    }
                }

                @Override
                public void onCancelled(@NonNull DatabaseError error) {
                    // Handle any errors
```

```
                }
            });
        }


}
```

Let's perform a more detailed analysis of statement coverage, branch coverage, and path coverage for the provided Java code.

1. Statement Coverage:

Statement coverage measures the percentage of executable code statements that have been executed during testing. It's a basic metric for assessing the completeness of your testing.

To calculate Statement Coverage, we count the number of executed statements divided by the total number of executable statements in the code.

Let's count the statements:

- Total Executable Statements: 33 (This includes lines with actual code to be executed)

To calculate Statement Coverage, we need to know how many of these statements have been executed during testing.

2. Branch Coverage:

Branch coverage measures the percentage of decision points (branches) in the code that have been executed. A decision point is typically an "if" statement or a switch-case statement.

To calculate Branch Coverage, we count the number of executed branches divided by the total number of branches in the code.

- Total Branches: Counting branches requires a detailed examination of conditional statements (e.g., if statements). It depends on the specific test cases and inputs used during testing. Without that information, I cannot provide an exact number for total branches.

3. Path Coverage:

Path coverage is a more advanced metric that considers all possible execution paths through the code. It provides insight into whether all logical paths have been tested.

Calculating Path Coverage is complex, as it depends on the number of conditional branches and nested conditions in the code. To determine Path Coverage, you need to analyze the logical paths and combinations of conditions tested during your testing. This involves creating a control flow graph and tracing which paths have been executed.

In practice, achieving 100% coverage for all these metrics is often challenging and may not be necessary, as it depends on project constraints and testing goals. The completeness of your testing should be based on the criticality of the code and the associated risks.

POSTLAB:

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

Here's a Java code snippet:

```
public int calculate(int x, int y) {
    int result;
    if (x > 0) {
        result = x + y;
    } else {
        result = x - y;
    }
    return result;
}
```

This code snippet contains two `if` statements. To achieve 100% statement coverage, we need to create test cases that ensure every statement is executed.

White-box test cases:

1. Test Case 1 (x > 0):
- Input: x = 3, y = 2
- Expected Output: 5
- Execution path: The code enters the if block, assigns `result = x + y`, and returns 5.

2. Test Case 2 (x <= 0):
- Input: x = -1, y = 5
- Expected Output: -6
- Execution path: The code enters the else block, assigns `result = x - y`, and returns -6.

3. Test Case 3 (x == 0):
- Input: x = 0, y = 10
- Expected Output: -10
- Execution path: The code enters the else block, assigns `result = x - y`, and returns -10.

These test cases ensure that every statement in the code snippet is executed. The code is tested with different conditions (x > 0, x <= 0, and x == 0) to cover all possible execution paths. By running these test cases, you achieve 100% statement coverage for the provided code snippet.

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 strengths and weaknesses. Let's compare and contrast these two testing methods in different testing scenarios:

White Box Testing:

1. Definition:
   - White box testing, also known as structural testing or glass-box testing, focuses on examining the internal structure, logic, and code of the software application. Testers have access to the source code, algorithms, and design, allowing them to create test cases based on this knowledge.

2. Strengths:
   - Thorough Coverage: White box testing can provide excellent statement, branch, and path coverage. It helps ensure that all code paths are tested, which is vital for complex and critical

applications.

- Defect Localization: It's effective in pinpointing the exact location of defects, making it easier for developers to identify and fix issues.
- Early Detection of Logic Errors: By analyzing code logic, it can identify logical errors and inconsistencies in the code early in the development process.

3. Weaknesses:

- Bias: Testers with access to the code may have biases that influence their test case design.
- Incomplete Testing: While it can provide excellent structural coverage, it might not always ensure adequate testing of user scenarios and requirements.
- Resource-Intensive: White box testing can be resource-intensive and time-consuming, especially for large codebases.

Black Box Testing:

1. Definition:

- Black box testing, also known as functional testing, focuses on the software's external behavior without any knowledge of the internal code. Testers design test cases based on the software's specifications, requirements, and expected functionality.

2. Strengths:

- Objective and Unbiased: Black box testing is objective and unbiased since testers do not need to know the internal code. This makes it suitable for independent verification.
- User-Centric: It aligns well with end-users' perspectives, ensuring that the software meets user expectations.
- Requirements Validation: It is effective for validating that the software complies with specified requirements and works as intended from the user's point of view.
- Efficiency: Black box testing can be more efficient for testing user scenarios and large-scale testing because it does not require knowledge of the code.

3. Weaknesses:

- Limited Structural Coverage: Black box testing may not provide comprehensive structural coverage. It may miss edge cases and code paths that are not explicitly covered in requirements.

- Difficulty in Defect Localization: It might not pinpoint the exact location of defects within the code, which can make debugging more challenging.
- Risk of Incomplete Testing: There's a risk that critical issues related to code logic may go unnoticed.

Scenarios for Each Approach:

- White Box Testing is Ideal When:
  - The software has complex algorithms that require in-depth testing.
  - You need to verify specific code paths and data flows.
  - There are strict regulatory or compliance requirements that demand comprehensive code coverage.

- Black Box Testing is Ideal When:
  - The focus is on validating user scenarios and requirements.
  - You need to assess the software from the end-user perspective.
  - Independent verification is necessary, and the internal code is proprietary or not accessible.

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 uncovering complex logic errors and security vulnerabilities. Here's an analysis of its impact on software quality and its ability to address these issues:

Uncovering Complex Logic Errors:

1. Code Path Coverage: White box testing, with its ability to analyze the internal code and data flows, can provide comprehensive code path coverage. This means that it can test various execution paths and branches in the code, including rare and complex scenarios. As a result, it is highly effective in uncovering complex logic errors that might be missed in black box testing.

2. Condition Testing: White box testing can assess different conditions, loops, and decision points in the code. It can identify issues like incorrect branching, unexpected loops, and logic discrepancies, which are often the root causes of complex logic errors.

3. Early Detection: By identifying complex logic errors early in the development process, white box testing helps prevent these issues from becoming critical defects in the production environment. Early detection reduces the cost and effort required for fixing these errors.

4. Regression Testing: White box testing is valuable for regression testing, ensuring that changes to the code do not introduce new complex logic errors while maintaining existing functionality.

Uncovering Security Vulnerabilities:

1. Source Code Analysis: White box testing has the advantage of examining the source code, which is critical for uncovering security vulnerabilities. Testers can identify insecure coding practices, such as input validation issues, authentication flaws, and insecure data storage.

2. Security Scanning Tools: White box testing can integrate security scanning tools that analyze the code for common security vulnerabilities like SQL injection, cross-site scripting (XSS), and access control problems.

3. Authentication and Authorization Testing: White box testing can verify that authentication and authorization mechanisms are correctly implemented and that access control rules are enforced, reducing the risk of unauthorized access to sensitive data.

4. Secure Data Handling: It can ensure that the application properly handles sensitive data, preventing data leakage and data integrity issues.

5. Protection Against Common Attacks: White box testing can evaluate how the software defends against common security threats, such as CSRF (Cross-Site Request Forgery), buffer overflows, and path traversal attacks.

6. Encryption and Data Protection: It can assess the proper use of encryption for data in transit and at rest, reducing the risk of data breaches.