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:	**9587**
Team Members:	**Sania Almeida**

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 to display various skin care products:

```
package com.example.skincare;


import android.app.AlertDialog;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.GridLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.google.android.material.floatingactionbutton.FloatingActionButton;
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 java.util.ArrayList;
import java.util.List;

public class product extends AppCompatActivity {

    RecyclerView recyclerView;
    List<Data> dataList;
    DatabaseReference databaseReference;
    ValueEventListener eventListener;
```

```
    Adapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_product);

        recyclerView = findViewById(R.id.recyclerView1);

        GridLayoutManager gridLayoutManager = new GridLayoutManager(product.this, 1);
        recyclerView.setLayoutManager(gridLayoutManager);

        AlertDialog.Builder builder = new AlertDialog.Builder(product.this);
        builder.setCancelable(false);
        builder.setView(R.layout.progress_layout);
        AlertDialog dailog = builder.create();
        dailog.show();

        dataList = new ArrayList<>();

        adapter = new Adapter(product.this, dataList);
        recyclerView.setAdapter(adapter);

        databaseReference = FirebaseDatabase.getInstance().getReference("Cleanser");
        dailog.show();

        eventListener = databaseReference.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                dataList.clear();
                for(DataSnapshot itemSnapshot: snapshot.getChildren()){
                    Data data = itemSnapshot.getValue(Data.class);
                    data.setKey(itemSnapshot.getKey());
                    dataList.add(data);
                }
                adapter.notifyDataSetChanged();
                dailog.dismiss();
            }

            @Override
            public void onCancelled(@NonNull DatabaseError error) {
                dailog.dismiss();

            }
        });


    }
}
```

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.

In the given code, the number of executable statements is as follows:

- Total number of executable statements: 31

To calculate Statement Coverage, you would need to run test cases and track how many of these 31 statements are executed. The coverage would be the ratio of executed statements to the total number of statements.

2. Branch Coverage:

- Branch coverage measures the percentage of decision points (branches) that have been executed during testing.

In the code, you have a conditional branch in the `onDataChange` method:

```
if (dataList.isEmpty()) {
   // executed when dataList is empty
} else {
   // executed when dataList is not empty
}
```

There's one branch in this conditional statement. To calculate Branch Coverage, you would need to determine if both branches (true and false) have been executed during testing.

3. Path Coverage:

- Path coverage measures the percentage of different paths through the code that have been executed during testing.

In the code, there are multiple paths through the code, depending on the data retrieved from the database and the flow of the program. Calculating path coverage requires considering various execution scenarios, including the execution of both `onDataChange` and `onCancelled` methods.

The code paths include:

- Successful execution of `onDataChange` (data retrieved from the database).
- Execution of `onCancelled` (error condition during database retrieval).

To calculate Path Coverage, you would need to ensure that all possible execution paths are tested, including both successful and error scenarios.

POSTLAB:

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

Here's a simple Python code snippet:

```python
def divide(a, b):
    if b == 0:
        return "Division by zero is not allowed"
    result = a / b
    return result
def is_positive(num):
    return num > 0
```

To achieve 100% statement coverage, we need to ensure that every line of code is executed at least once. Here are the test cases:

Test Case 1: Valid division
```python
a = 10
b = 2
result = divide(a, b)
assert result == 5  # Check if the result is correct
```

Test Case 2: Division by zero
```python
a = 5
b = 0
result = divide(a, b)
assert result == "Division by zero is not allowed"  # Check the error message
```

Test Case 3: Check if a number is positive
```python
num = 7
assert is_positive(num)  # Check if the function correctly identifies a positive number
```

Test Case 4: Check if a number is not positive
```python
num = -3
assert not is_positive(num)  # Check if the function correctly identifies a non-positive number
```
By executing these four test cases, you will achieve 100% statement coverage for the provided code snippet. Each line of code is executed at least once in these tests. Keep in mind that this is a

simplified example, and real-world code may require more comprehensive testing to achieve complete coverage and handle different scenarios and edge cases.

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.