Department of Computer Engineering

Academic Term: First Term 2023-24 Class: T.E

/Computer Sem – V / Software Engineering

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)	

Practical No:	**9**
Title:	**Designing Test Cases for Performing White Box Testing**
Date of Performance:	**14/10/2023**
Roll No:	**9547**
Team Members:	**Ryan D'Mello, Leslie D silva, Pradyumnaa Kadam**

Rubrics for Evaluation:

Signature of the Teacher:

Department of Computer Engineering

Academic Term: First Term 2022-23 Class: T.E

/Computer Sem – V / Software Engineering

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.

White-box testing is a method of software testing that examines the internal structures or workings of an application. When it comes to a login authentication using django, whitebox testing can be performed to ensure that the code is functioning as expected.
Let's consider a simplified example of a login authentication system using Django. The system is designed to verify user details and provide secure access. Here's an example of how you can conduct white-box testing for the system:

Code:

```
class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):        if
not email:
            raise ValueError('The Email field must be set')
email = self.normalize_email(email)        user =
self.model(email=email, **extra_fields)
user.set_password(password)        user.save(using=self._db)
return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
extra_fields.setdefault('is_superuser', True)

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')        if
extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')

        return self.create_user(email, password, **extra_fields)

class CustomUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)     is_active =
models.BooleanField(default=True)     is_staff =
models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    def __str__(self):        return
self.email
// the Django model for user login
```

```
def login_view(request):     if
request.method == 'POST':
        form = CustomLoginForm(request, data=request.POST)        if
form.is_valid():
```

```
        login(request, form.get_user())            return redirect('success_url')  # Replace 'success_url'
with your desired success URL.     else:
      form = CustomLoginForm()     return render(request,
'login.html', {'form': form})
```

//Django view for login form

White-Box Testing Steps:

Code Review: Review the source code to understand its structure, including variables, functions, and conditional statements.

Static Analysis: Check for any potential issues in the code without actually running it. Ensure there are no syntax errors, variable conflicts, or other coding issues.

Unit Testing:

Test the LoginForm and POST request functions to ensure that they are functioning properly.
Test the login function with different input values for email and password, including dissimilar values, to confirm that the system responds accordingly.
Path Testing: Ensure that all possible paths through the code are tested, including the if-else conditions.

Boundary Testing: Test the system with values on the threshold (500 in this case) to verify if the alarm is triggered appropriately.

Integration Testing: Check if the hardware and software components are working together as expected. Simulate the gas sensor input and verify that the alarm is activated and deactivated correctly.

Performance Testing: Run the code for an extended period to check for any memory leaks or performance issues.

Error Handling Testing: Intentionally introduce errors or unexpected input to check if the system can handle such situations gracefully.

TEST CASE ID	TEST DESCRIPTION	TEST CASE	TEST STEPS	EXP. RESULT	ACTUAL RESULT	STATUS
TC_01	Testing LoginForm function	submission of form	set form variables and event handlers	db updated with values	db reflects changes in realtime	PASS
TC_02	Testing POST request function	sending of form data	set POST request function to send data to db	variable data sent	variable data received/validated	PASS
TC_03	Testing redirect function	redirect after login	finish entering details and click on submit	new page is loaded	redirected to new page	PASS

Postlab:

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

To achieve 100% statement coverage in white-box testing, you need to ensure that every statement in the code is executed at least once. Here's an example of a code snippet and the corresponding white-box test cases to achieve 100% statement coverage:

Python code:

```
def divide(a, b):
    result = 0
    if b != 0:
      result = a / b
    return result
```

In this code snippet, there are three statements to be covered: two lines in the function body and one line in the if statement. Here are test cases to achieve 100% statement coverage:

Test Case 1:

- Inputs: a = 10, b = 2 - Execution:
- a is 10, b is 2 (line 1)
- b is not equal to 0 (line 2, entering if statement)
- Calculation: result = 10 / 2 (line 3)
- Return result (line 4)

Test Case 2: -

Inputs: a = 0, b = 5 -

Execution:

- a is 0, b is 5 (line 1)
- b is not equal to 0 (line 2, entering if statement)
- Calculation: result = 0 / 5 (line 3)
- Return result (line 4)

Test Case 3: -

Inputs: a = 8, b = 0 -

Execution:

- a is 8, b is 0 (line 1)
- b is equal to 0 (line 2, not entering if statement)
- Return result (line 4)

These three test cases cover all the statements in the code snippet, ensuring 100% statement coverage.

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 testing methodologies, each with its own strengths and weaknesses. Let's compare and contrast these two approaches, highlighting their characteristics in different testing scenarios:

White-Box Testing:

1. **Focus:**

- **Internal Structure:** White-box testing is focused on the internal structure, code logic, and implementation details of the software.

2. **Tester Knowledge:**

- **High Tester Expertise:** Testers need a deep understanding of the code, algorithms, and system architecture.

3. **Test Design:**

- **Tests Designed with Code Knowledge:** Test cases are designed with specific knowledge of the codebase, which allows testers to target specific paths, conditions, and code branches.

4. **Strengths:**

- **Thorough Coverage:** White-box testing can achieve high code coverage, including statement, branch, and path coverage.

- **Early Detection of Bugs:** It's effective at catching logic and coding errors early in the development process.

- **Optimal Performance:** Performance-related issues can be identified through detailed code analysis.

5. **Weaknesses:**

- **Limited Real-World Perspective:** Testers may overlook real-world use cases and scenarios.

- **Bias and Assumptions:** The testing process may be influenced by the tester's preconceptions.

- **Inability to Validate External Factors:** It may not effectively test integration points or external dependencies.

Black-Box Testing:

1. **Focus:**

- **External Behavior:** Black-box testing concentrates on the software's external behavior, inputs, and expected outputs.

2. **Tester Knowledge:**

- **Low Tester Expertise:** Testers don't need detailed knowledge of the internal code or system architecture.

3. **Test Design:**

- **Tests Designed from User's Perspective:** Test cases are designed based on user requirements and specifications, without considering internal code.

4. **Strengths:**

- **Real-World Validation:** Black-box testing simulates real-world user interactions, making it effective for validating user scenarios and requirements.

- **Independence from Implementation:** Testers can evaluate the software without any bias or knowledge of how it's implemented.

- **Usability Testing:** It's suitable for assessing usability, user interface, and overall user experience.

5. **Weaknesses:**

- **Incomplete Code Coverage:** Black-box testing may not guarantee 100% code coverage, and some code paths might remain untested.

- **Limited Depth:** It may not uncover deep-seated logic or implementation issues.

- **Limited Effectiveness for Complex Algorithms:** Testing complex mathematical or algorithmic components can be challenging without understanding the underlying code.

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 enhancing software quality by uncovering complex logic errors and security vulnerabilities. Here's an analysis of its impact on software quality:

1. **Identification of Complex Logic Errors:**

- **Deep Code Analysis:** White-box testing involves a thorough examination of the internal code structure, allowing testers to identify complex logic errors that might be challenging to detect through black-box testing alone.

- **Boundary Conditions:** Testers can explore edge cases, boundary conditions, and unusual scenarios that can reveal hidden logic flaws. This helps ensure the software behaves correctly in all situations.
- **Data Flow Analysis:** By tracing the flow of data and control within the code, white-box testing can uncover issues such as data corruption, incorrect variable manipulation, and unexpected code paths.

2. **Early Detection of Bugs:**

- White-box testing can be integrated into the development process, catching coding errors, bugs, and logical issues early in the software development lifecycle. This reduces the cost and effort required for bug fixing at later stages.

3. **High Code Coverage:**

- White-box testing strives to achieve high code coverage, including statement, branch, and path coverage. This ensures that most parts of the code are tested, reducing the chances of undiscovered defects.

4. **Security Vulnerability Discovery:**

- White-box testing is instrumental in identifying security vulnerabilities such as injection attacks, privilege escalation, and authentication flaws. Testers can explore how the software handles various inputs and data manipulations, which is essential for security testing.

- **Source Code Analysis:** White-box testing tools can analyze the source code for security vulnerabilities and compliance with security best practices. This can help identify issues like code injection, buffer overflows, and sensitive data exposure.

5. **Improved Code Quality:**

- As white-box testing uncovers and rectifies complex logic errors and security vulnerabilities, it leads to improved code quality. This, in turn, contributes to a more robust and reliable software product.

6. **Regulatory Compliance:**

- Many industries have regulatory requirements regarding security and code quality. White-box testing can help ensure compliance with these standards by systematically verifying that security measures are in place and effective.

7. **Confidence in Software:**

- By thoroughly examining the internal workings of the software, white-box testing provides stakeholders with greater confidence in the software's reliability and security. This is particularly important for critical applications and systems.

Despite its numerous benefits, it's essential to acknowledge that white-box testing has its limitations, such as the inability to replicate all real-world scenarios and the potential for

biased testing based on the tester's knowledge. For comprehensive testing, a combination of white-box, black-box, and gray-box testing methods is often recommended. This ensures that both the internal structure and external behavior of the software are thoroughly evaluated, leading to higher software quality.