Experiment No. - 9

Name- Bhavesh Kewalramani

Roll No.- A-25

Section- A

Semester- 6th

Shift- 1st

Aim:

To perform White-Box Testing to test the functionalities and perform Black-Box Testing on a test case template for the system.

Theory:

Black Box Testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

The above Black-Box can be any software system you want to test. For Example, an operating system like Windows, a website like Google, a database like Oracle or even your own custom application.

How to do BlackBox Testing

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.

- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones –

- **Functional testing** – This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** – Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Tools used for Black Box Testing:

Tools used for Black box testing largely depends on the type of black box testing you are doing.

- For Functional/ Regression Tests you can use – QTP, Selenium
- For Non-Functional Tests, you can use – LoadRunner, Jmeter

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Comparison of Black Box and White Box Testing:

Which to choose ???

Black Box Testing	White Box Testing
the main focus of black box testing is on the validation of your functional requirements.	White Box Testing (Unit Testing) validates internal structure and working of your software code
Black box testing gives abstraction from code and focuses on testing effort on the software system behavior.	To conduct White Box Testing, knowledge of underlying programming language is essential. Current day software systems use a variety of programming languages and technologies and its not possible to know all of them.
Black box testing facilitates testing communication amongst modules	White box testing does not facilitate testing communication amongst modules

Black Box Testing and Software Development Life Cycle (SDLC)

Black box testing has its own life cycle called Software Testing Life Cycle (STLC) and it is relative to every stage of Software Development Life Cycle of Software Engineering.

- **Requirement** – This is the initial stage of SDLC and in this stage, a requirement is gathered. Software testers also take part in this stage.
- **Test Planning & Analysis** – Testing Types applicable to the project are determined. A Test Plan is created which determines possible project risks and their mitigation.
- **Design** – In this stage Test cases/scripts are created on the basis of software requirement documents
- **Test Execution**– In this stage Test Cases prepared are executed. Bugs if any are fixed and re-tested.

White Box Testing

White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.

It is one of two parts of the Box Testing approach to software testing. Its counterpart, Blackbox testing, involves testing from an external or end-user type perspective. On the other hand, White box testing in software engineering is based on the inner workings of an application and revolves around internal testing.

The term "WhiteBox" was used because of the see-through box concept. The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

What do you verify in White Box Testing?

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

The testing can be done at system, integration and unit levels of software development. One of the basic goals of whitebox testing is to verify a working flow for an application. It involves testing a series of predefined inputs against expected or desired outputs so that when a specific input does not result in the expected output, you have encountered a bug.

How do you perform White Box Testing?

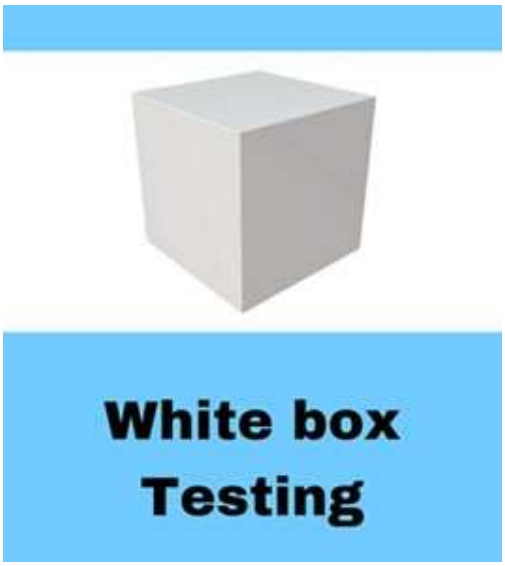
To give you a simplified explanation of white box testing, we have divided it into **two basic steps**. This is what testers do when testing an application using the white box testing technique:

STEP 1) UNDERSTAND THE SOURCE CODE

The first thing a tester will often do is learn and understand the source code of the application. Since white box testing involves the testing of the inner workings of an application, the tester must be very knowledgeable in the programming languages used in the applications they are testing. Also, the testing person must be highly aware of secure coding practices. Security is often one of the primary objectives of testing software. The tester should be able to find security issues and prevent attacks from hackers and naive users who might inject malicious code into the application either knowingly or unknowingly.

Step 2) CREATE TEST CASES AND EXECUTE

The second basic step to white box testing involves testing the application's source code for proper flow and structure. One way is by writing more code to test the application's source code. The tester will develop little tests for each process or series of processes in the application. This method requires that the tester must have intimate knowledge of the code and is often done by the developer. Other methods include Manual Testing, trial, and error testing and the use of testing tools as we will explain further on in this article.



WhiteBox Testing Example

Consider the following piece of code

```
Printme (int a, int b) {                    ----------  Printme is a function
   int result = a+ b;
   If (result> 0)
            Print ("Positive", result)
   Else
            Print ("Negative", result)
   }                                  ----------   End of the source code
```

The goal of WhiteBox testing in software engineering is to verify all the decision branches, loops, statements in the code.

To exercise the statements in the above white box testing example, WhiteBox test cases would be

- A = 1, B = 1
- A = -1, B = -3

White Box Testing Techniques

A major White box testing technique is Code Coverage analysis. Code Coverage analysis eliminates gaps in a Test Case suite. It identifies areas of a program that are not exercised by

a set of test cases. Once gaps are identified, you create test cases to verify untested parts of the code, thereby increasing the quality of the software product

There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques a box tester can use:

Statement Coverage:- This technique requires every possible statement in the code to be tested at least once during the testing process of software engineering.

Branch Coverage – This technique checks every possible path (if-else and other conditional loops) of a software application.

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code. **Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.**

Following are important WhiteBox Testing Techniques:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Multiple Condition Coverage
- Finite State Machine Coverage
- Path Coverage
- Control flow testing
- Data flow testing

Types of White Box Testing

White box testing encompasses several testing types used to evaluate the usability of an application, block of code or specific software package. There are listed below —

- **Unit Testing:** It is often the first type of testing done on an application. Unit Testing is performed on each unit or block of code as it is developed. Unit Testing is essentially done by the programmer. As a software developer, you develop a few lines of code, a single function or an object and test it to make sure it works before continuing Unit Testing helps identify a majority of bugs, early in the software development lifecycle. Bugs identified in this stage are cheaper and easy to fix.
- **Testing for Memory Leaks**: Memory leaks are leading causes of slower running applications. A QA specialist who is experienced at detecting memory leaks is essential in cases where you have a slow running software application.

Apart from above, a few testing types are part of both black box and white box testing. They are listed as below

- **White Box Penetration Testing:** In this testing, the tester/developer has full information of the application's source code, detailed network information, IP

addresses involved and all server information the application runs on. The aim is to attack the code from several angles to expose security threats
- **White Box Mutation Testing**: Mutation testing is often used to discover the best coding techniques to use for expanding a software solution.

White Box Testing Tools

Below is a list of top white box testing tools.

- Parasoft Jtest
- EclEmma
- NUnit
- PyUnit
- HTMLUnit
- CppUnit

Advantages of White Box Testing

- Code optimization by finding hidden errors.
- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if GUI is not available.

Disadvantages of WhiteBox Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- White box testing requires professional resources, with a detailed understanding of programming and implementation.
- White-box testing is time-consuming, bigger programming applications take the time to test fully.

Ending Notes:

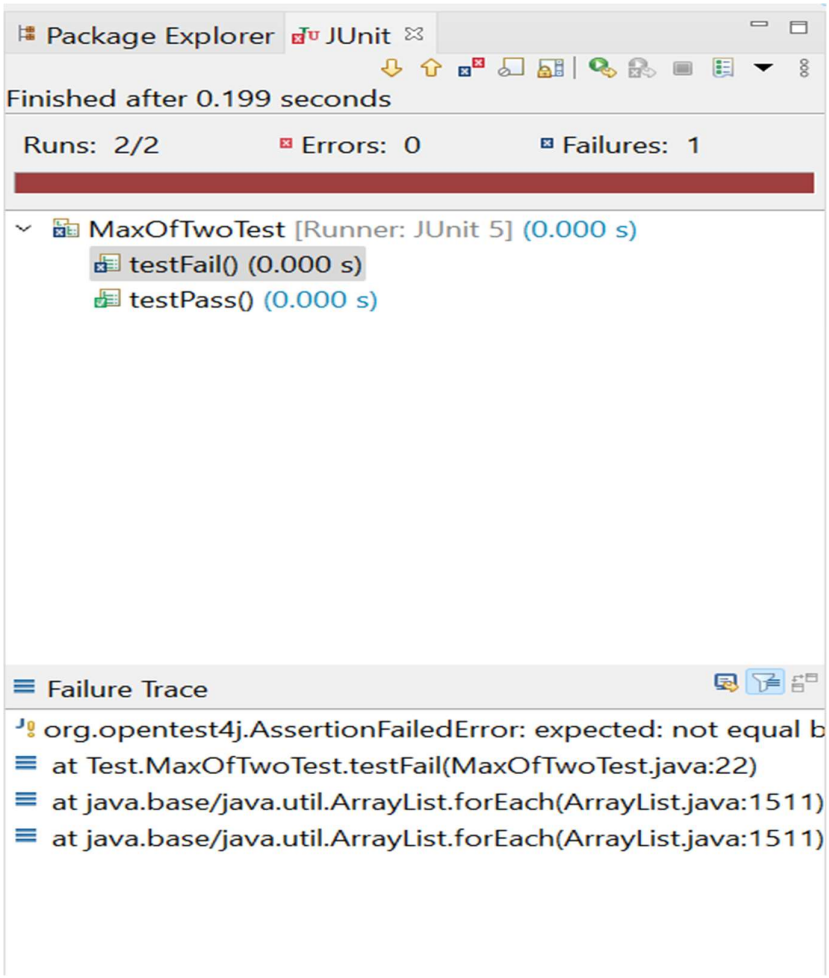
- White box testing can be quite complex. The complexity involved has a lot to do with the application being tested. A small application that performs a single simple operation could be white box tested in few minutes, while larger programming applications take days, weeks and even longer to fully test.
- White box testing in software testing should be done on a software application as it is being developed after it is written and again after each modification

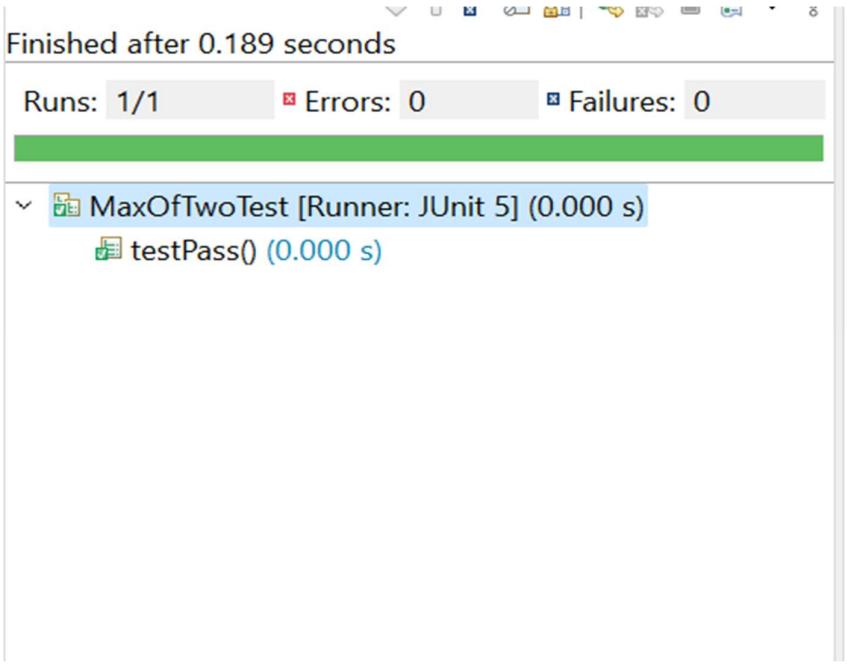
White Box Testing

1. To find Max of two no.s.

```
package Test;

import java.util.Scanner;

public class MaxOfTwo {

	public static int max(int x,int y) {
		if(x>y) {
			return x;
		}
		else {
			return y;
		}
	}

}
```

```
package Test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class MaxOfTwoTest {

	@Test
	void testPass() {
		MaxOfTwo obj1 = new MaxOfTwo();
		int input_f = obj1.max(5, 3);
		assertEquals(5,input_f);
		input_f = obj1.max(2,3);
		assertEquals(3,input_f);
	}

//	@Test
//	void testFail(){
//		MaxOfTwo obj1 = new MaxOfTwo();
//		int input_f = obj1.max(5, 5);
//		assertNotEquals(5,input_f);
//	}

}
```


Package Explorer
JUnit
Finished after 0.199 seconds
Runs: 2/2
Errors: 0
Failures: 1
MaxOfTwoTest [Runner: JUnit 5] (0.000 s)
testFail() (0.000 s)
testPass() (0.000 s)
Failure Trace
org.opentest4j.AssertionFailedError: expected: not equal b
at Test.MaxOfTwoTest.testFail(MaxOfTwoTest.java:22)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)


Finished after 0.189 seconds
Runs: 1/1
Errors: 0
Failures: 0
MaxOfTwoTest [Runner: JUnit 5] (0.000 s)
testPass() (0.000 s)

2. To build a Calculator(Add,Subs,Mul,Div)

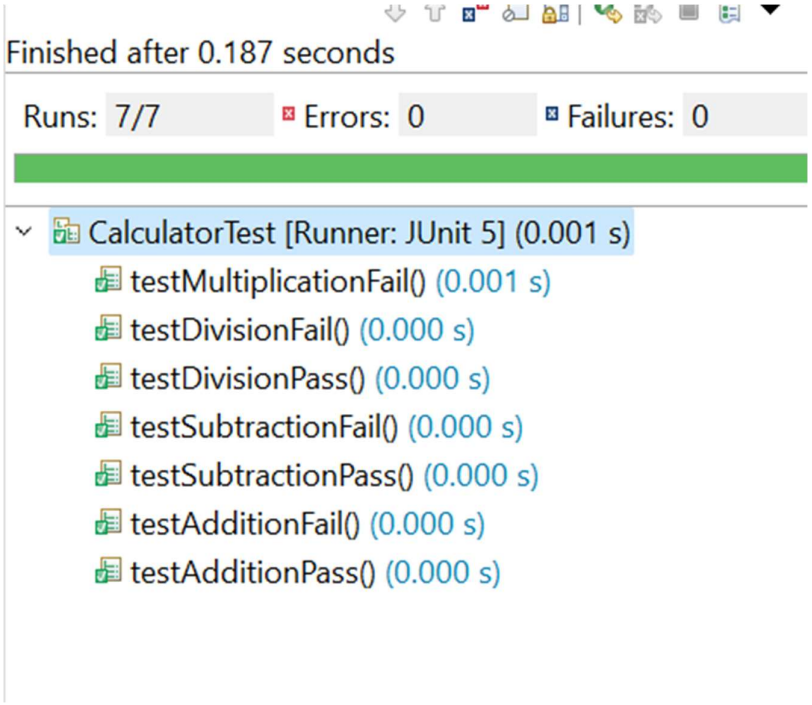
```
package Test;

public class Calculator {
	public static int addition(int num1, int num2) {
		return num1 + num2;
	   }

	   public static int subtraction(int num1, int num2) {
	      return num1 - num2;
	   }

	   public static int multiplication(int num1, int num2) {
	      return num1 * num2;
	   }

	   public static int division(int num1, int num2) {
	      if (num2 == 0) {
	         throw new IllegalArgumentException("Cannot divide by 0!");
	      }
	      return num1 / num2;
	   }
}
```

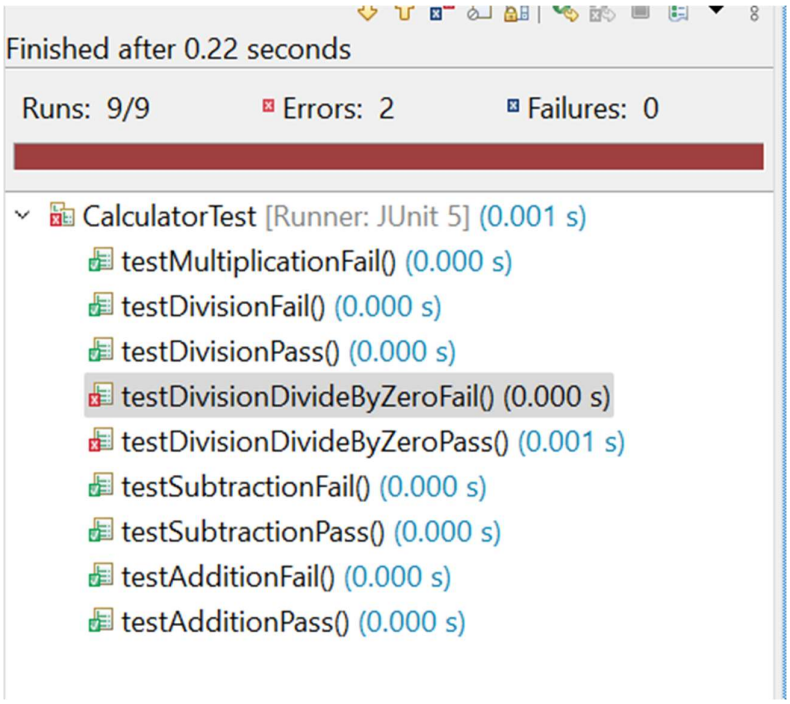
```
package Test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class CalculatorTest {

	@Test
	public void testAdditionPass() {
	  // assertEquals(String message, long expected, long actual)
	  assertEquals("error in addition()",  3, Calculator.addition(1, 2));
	  assertEquals("error in addition()", -3, Calculator.addition(-1, -2));
```

```
    assertEquals("error in addition()",  9, Calculator.addition(9, 0));
  }

  @Test
  public void testAdditionFail() {
    // assertNotEquals(String message, long expected, long actual)
    assertNotEquals("error in addition()", 0, Calculator.addition(1, 2));
  }

  @Test
  public void testSubtractionPass() {
    assertEquals("error in subtraction()",  1, Calculator.subtraction(2, 1));
    assertEquals("error in subtraction()", -1, Calculator.subtraction(-2, -1));
    assertEquals("error in subtraction()",  0, Calculator.subtraction(2, 2));
  }

  @Test
  public void testSubtractionFail() {
    assertNotEquals("error in subtraction()", 0, Calculator.subtraction(2, 1));
  }

  public void testMultiplicationPass() {
        // assertEquals(String message, long expected, long actual)
        assertEquals("error in multiplication()", 2, Calculator.multiplication(1, 2));
        assertEquals("error in multiplication()", 2, Calculator.multiplication(-1, -
2));
        assertEquals("error in multiplication()", 0, Calculator.multiplication(9, 0));
      }

      @Test
      public void testMultiplicationFail() {
        // assertNotEquals(String message, long expected, long actual)
```

```
        assertNotEquals("error in multiplication()", 0, Calculator.multiplication(1,
2));
    }

    @Test
    public void testDivisionPass() {
      assertEquals("error in division()", 2, Calculator.division(2, 1));
      assertEquals("error in division()", 2, Calculator.division(-2, -1));
      assertEquals("error in division()", 1, Calculator.division(2, 2));
    }

    @Test
    public void testDivisionFail() {
      assertNotEquals("error in division()", 0, Calculator.division(2, 1));
    }

//  @Test
//  public void testDivisionDivideByZeroPass() {
//      assertEquals("error in division()", new
IllegalArgumentException("Cannot divide by 0!"), Calculator.division(2, 0));
//      assertEquals("error in division()", new
IllegalArgumentException("Cannot divide by 0!"), Calculator.division(-2, 0));
//      assertEquals("error in division()", new
IllegalArgumentException("Cannot divide by 0!"), Calculator.division(3, 0));
//  }
//
//  @Test
//  public void testDivisionDivideByZeroFail() {
//      assertNotEquals("error in division()", 5, Calculator.division(2, 0));
//  }
```

}


Finished after 0.187 seconds
Runs: 7/7
Errors: 0
Failures: 0
CalculatorTest [Runner: JUnit 5] (0.001 s)
testMultiplicationFail() (0.001 s)
testDivisionFail() (0.000 s)
testDivisionPass() (0.000 s)
testSubtractionFail() (0.000 s)
testSubtractionPass() (0.000 s)
testAdditionFail() (0.000 s)
testAdditionPass() (0.000 s)


Finished after 0.22 seconds
Runs: 9/9
Errors: 2
Failures: 0
CalculatorTest [Runner: JUnit 5] (0.001 s)
testMultiplicationFail() (0.000 s)
testDivisionFail() (0.000 s)
testDivisionPass() (0.000 s)
testDivisionDivideByZeroFail() (0.000 s)
testDivisionDivideByZeroPass() (0.001 s)
testSubtractionFail() (0.000 s)
testSubtractionPass() (0.000 s)
testAdditionFail() (0.000 s)
testAdditionPass() (0.000 s)

Black Box Testing

1. Website Testing - eg. Flipkart.com
   action, input,broweser,expected outcome, actual output, result

Website Testing						
Website Name:	**Amazon.com**					

* Write Test cases to purchase an item

Sr. No.	Actions	Input	Browser	Expected Output	Actual Output	Result
1	Launch Website and check default products	https://www.amazon.com/	Gooogle Chrome	Website should launch, Default products appear	Launch Successful, Most searched or sold products appeared	Pass
2	Click on the Login Button	Empty Fields	Gooogle Chrome	Invalid request	Displaying Invalid Request Message	Pass
3	Click on the Login Button	Filled Fields	Gooogle Chrome	Registered	Registered	Pass
4	Search Product	Fill product names	Google Chrome	Related Products appear	Similar Products Appear	Pass
5	Place order without Login	Select and add Products to cart	Google Chrome	Proceed to Login Page	Proceed To Login	Pass
6	Place order with Login	Login, select and add Products to Cart	Google Chrome	Proceed to Checkout	Proceed To Checkout	Pass
7	Fill Bank and Shipping details and Place order	Fill Bank and shipping details and verify it	Google Chrome	Place order	Place order	Pass

2. Tool Testing - NetBeans IDE
   action, input, expected outcome, actual output, result

Tool Testing					
Tool Name: Eclipse IDE					

* Write Test cases to create and run the Project

Sr. No.	Actions	Input	Expected Output	Actual Output	Result
1	Launch Eclipse IDE	Eclipse workspace	Show existing projects in workspace	shows existig projects in workspace	Pass
2	Create Project	Project Name	Project Created Successfully	Project Created Successfully	Pass
3	Create Package in Project	Package Name	Package Created Successfully	Package Created Successfully	Pass
4	Create Class	Class Name	Class Created Successfully	Class Created Successfully	Pass
5	Add code to add two numbers	java code	Compilation Successful	Compilation Successful	Pass
6	Enter two numbers	integers	Addition Result	Addition Result	Pass
7	Add two numbers	2,5	7	7	Pass

3.
1. Code Testing-
2. Functional Coverage Test
3. Special Value Test
4. Boundary Value Test

Code-

```
# def add(a,b):
#     return a+b
#
# a,b=[int(x) for x in input().split()]
# string="Hello"
# c=a+b
# if c<10:
#     print("Value of c is : ",c)
# else:
#     print("Sorry")
```

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    L = [0] * (n1)
    R = [0] * (n2)

    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    i = 0
    j = 0
    k = l

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1



def mergeSort(arr, l, r):
    if l < r:
        m = l + (r - l) // 2

        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)


arr=[]
n = int(input())
for i in range(n):
    arr.append(int(input()))
# print("Given array is")
# for i in range(n):
#     print("%d" % arr[i], end=" ")

mergeSort(arr, 0, n - 1)
# print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i], end=" ")
```

Representative input data

Test #	Test Description	Test Data	Expected Result	Actual Result	Pass/Fail
1	Normal Numbers	5 3 2 1 4 2	1 2 2 3 4	1 2 2 3 4	Pass
2	Zeros	4 0 0 0 1	0 0 0 1	0 0 0 1	Pass
3	Normal Numbers	4 3 2 1 4	1 2 3 4	1 2 3 4	Pass

Boundary Value test

Test #	Test Description	Test Data	Expected Result	Actual Result	Pass/Fail
1	Number less than boundary	4 -999 -1000 - 998 - 997	-1000 -999 -998 -997	-1000 -999 -998 -997	Pass
2	Number having exact boundary value	4 0 0 1 0	0 0 0 1	0 0 0 1	Pass
3	Number greater than boundary	999 1000 998 997	997 998 999 1000	997 999 998 1000	Fail

Functional Coverage test

Test #	Test Description	Test Data	Expected Result	Actual Result	Pass Fail
1	String with number	"1" "2"	Invalid	Invalid	Pass
2	String and an integer	"21", 20	Invalid	Invalid	Pass
3	Negative number	-21 -20 -99	-99 -21 -20	-99 -21 -20	Pass

Conclusion:

White Box helps in the testing of the application at unit, integration and system level. It also helps to verification of logical conditions and its values in the form of correct/false result values. It also helps to verify all independent paths within a module by using different techniques that are available under these testing techniques. It helps to detect loops are optimum, and there are no extra looping works under a specified area or functionality. The main benefit of this is that it helps to find the bugs at different levels and the errors named as a code syntax error, logical errors in code; Data flow errors and conditional errors. It works on an internal level and also helps to find bugs at a profound level

Black box testing helps to find the gaps in functionality, usability, and other features. This form of testing gives an overview of software performance and its output. It improves software quality and reduces the time to market. This form of testing mitigates the risk of software failures at the user's end.