1. Examine an ATM System: Select an ATM system, thoroughly review

its system specifications, and identify and document the various software

defects present.

class ATM {

private int pin = 1234;

private double balance = 500;

private boolean auth = false;

public boolean login(int p) {

auth = (p == pin);

return auth;

}

public double getBalance() { return balance; }

public boolean withdraw(double amt) {

if (!auth) { System.out.println("Error: Invalid PIN."); return false; }

if (amt <= 0 || amt > balance) { System.out.println("Error: Invalid amount."); return false; }

balance -= amt;

System.out.println("Dispensed $" + amt);

System.out.println("Receipt Printed");

System.out.println("Updated Balance: $" + balance);

return true;

}

}

public class ATMTest {

public static void main(String[] args) {

ATM atm = new ATM();

System.out.println("Input: User selects withdrawal option, enters PIN=1234, requests $100");

if (atm.login(1234)) {

atm.withdraw(100);

} else {

System.out.println("Invalid PIN");

}

}

}

1. Case Study on ARMS Portals: Analyzing Requirements and Developing Positive Test Scenarios

class ARMSPortal {

private String user = "testUser";

private String pass = "testPass";

public boolean login(String u, String p) {

return u.equals(user) && p.equals(pass);

}

public String dashboard() {

return "Welcome to Dashboard";

}

}

public class ARMSPortalTest {

public static void main(String[] args) {

ARMSPortal portal = new ARMSPortal();

System.out.println("Input: username=testUser, password=testPass");

if (portal.login("testUser", "testPass")) {

System.out.println("Output: " + portal.dashboard());

} else {

System.out.println("Output: Invalid Login");

}

}

}

1. Case Study on ARMS Portals: Analyzing Requirements and Developing Negative Test

Scenarios

class ARMSPortal {

private String user = "testUser";

private String pass = "testPass";

public boolean login(String u, String p) {

return u.equals(user) && p.equals(pass);

}

}

public class ARMSPortalNegativeTest {

public static void main(String[] args) {

ARMSPortal portal = new ARMSPortal();

System.out.println("Input: username=wrongUser, password=wrongPass");

if (portal.login("wrongUser", "wrongPass")) {

System.out.println("Output: Login Success (Error in handling!)");

} else {

System.out.println("Output: Invalid username or password");

}

}

}

1. Case Study on e-Commerce Applications: Analyzing Requirements and Creating Positive Test Scenarios

class ECommerceApp {

private String cart = "";

public void browse(String product) {

System.out.println("Product Selected: " + product);

cart = product;

}

public void checkout() {

if (!cart.isEmpty()) System.out.println("Proceeding to Checkout with " + cart);

}

public boolean payment(String card) {

return card.equals("validCard");

}

public void confirmOrder() {

System.out.println("Order Confirmed and Added to History");

}

}

public class ECommercePositiveTest {

public static void main(String[] args) {

ECommerceApp app = new ECommerceApp();

System.out.println("Input: User selects product, adds to cart, checkout, enters valid payment, confirms order");

app.browse("Laptop");

app.checkout();

if (app.payment("validCard")) {

System.out.println("Output: Payment Successful");

app.confirmOrder();

} else {

System.out.println("Output: Payment Failed");

}

}

}

1. Case Study on e-Commerce Applications: Analyzing Requirements and Creating Negative Test Scenarios

class ECommerceApp {

private String cart = "";

public void browse(String product) {

System.out.println("Product Selected: " + product);

cart = product;

}

public void checkout() {

if (cart.isEmpty()) {

System.out.println("Output: Please add items to your cart");

} else {

System.out.println("Proceeding to Checkout with " + cart);

}

}

public boolean payment(String card) {

if (!card.equals("validCard")) {

System.out.println("Output: Invalid payment details. Please try again.");

return false;

}

return true;

}

}

public class ECommerceNegativeTest {

public static void main(String[] args) {

ECommerceApp app = new ECommerceApp();

System.out.println("Input: User tries checkout without adding item to cart");

app.checkout();

System.out.println("\nInput: User adds product but enters invalid payment info");

app.browse("Mobile");

app.checkout();

if (!app.payment("wrongCard")) {

System.out.println("Transaction Failed");

}

}

}

1. Case Study on Food Delivery Applications Analyzing Requirements and Creating Positive Test Scenarios

class FoodDeliveryApp {

private String cart = "";

public void browseMenu(String item) {

System.out.println("Menu Item Selected: " + item);

cart = item;

}

public void checkout() {

if (!cart.isEmpty()) {

System.out.println("Proceeding to Checkout with " + cart);

}

}

public boolean payment(String card) {

return card.equals("validCard");

}

public void confirmOrder() {

System.out.println("Order Confirmed | Estimated Delivery: 30 minutes");

}

}

public class FoodDeliveryPositiveTest {

public static void main(String[] args) {

FoodDeliveryApp app = new FoodDeliveryApp();

System.out.println("Input: User browses menu, selects items, adds to cart, checkout, enters valid payment, confirms order");

app.browseMenu("Pizza");

app.checkout();

if (app.payment("validCard")) {

System.out.println("Output: Payment Successful");

app.confirmOrder();

} else {

System.out.println("Output: Payment Failed");

}

}

}

1. Case Study on Food Delivery Applications Analyzing Requirements and Creating Negative Test Scenarios

class FoodDeliveryApp {

private String cart = "";

public void checkout() {

if (cart.isEmpty()) System.out.println("Output: Your cart is empty. Please add items.");

}

public boolean payment(String card) {

if (!card.equals("validCard")) {

System.out.println("Output: Invalid payment method. Please check your card details.");

return false;

}

return true;

}

}

public class FoodDeliveryNegativeTest {

public static void main(String[] args) {

FoodDeliveryApp app = new FoodDeliveryApp();

System.out.println("Input: Checkout without selecting items");

app.checkout();

System.out.println("\nInput: User provides invalid card");

if (!app.payment("wrongCard")) System.out.println("Transaction Failed");

}

}

1. Case Study on Banking Applications: Analyzing Requirements and Developing Positive Test Scenarios

class BankingApp {

private String user = "admin", pass = "1234";

private double balance = 1000;

public boolean login(String u, String p) { return u.equals(user) && p.equals(pass); }

public boolean transfer(double amt) {

if (amt <= balance) {

balance -= amt;

System.out.println("Output: Transfer Successful | New Balance: " + balance);

return true;

}

return false;

}

}

public class BankingPositiveTest {

public static void main(String[] args) {

BankingApp app = new BankingApp();

System.out.println("Input: Valid login and transfer 500");

if (app.login("admin", "1234")) app.transfer(500);

else System.out.println("Output: Invalid Login");

}

}

1. Case Study on Banking Applications: Analyzing Requirements and Developing Negative Test Scenarios

class BankingApp {

private String user = "admin", pass = "1234";

private double balance = 500;

public boolean login(String u, String p) { return u.equals(user) && p.equals(pass); }

public boolean transfer(double amt) {

if (amt > balance) {

System.out.println("Output: Insufficient funds. Please check your balance.");

return false;

}

return true;

}

}

public class BankingNegativeTest {

public static void main(String[] args) {

BankingApp app = new BankingApp();

System.out.println("Input: Invalid login attempt");

if (!app.login("admin", "wrong")) System.out.println("Output: Invalid username or password");

System.out.println("\nInput: Transfer exceeding balance");

if (!app.transfer(1000)) System.out.println("Transaction Failed");

}

}

1. Test Plan Development for Library Management System

class LibrarySystem {

private String user = "admin", pass = "admin123";

public boolean login(String u, String p) { return u.equals(user) && p.equals(pass); }

}

public class LMSTest {

public static void main(String[] args) {

LibrarySystem lms = new LibrarySystem();

System.out.println("Input 1: username=admin, password=admin123");

if (lms.login("admin", "admin123")) System.out.println("Output: Login Successful");

System.out.println("\nInput 2: username=admin, password=wrongpassword");

if (!lms.login("admin", "wrongpassword")) System.out.println("Output: Invalid username or password");

}

}

11. JUnit Test for Reversing a Word

Aim:

To write a JUnit test that verifies the functionality of a method that reverses a given word.

The JUnit test will use assert statements to check if the word is reversed correctly.

Algorithm:

1. Step 1: Create a method to reverse the input word.

o Use a StringBuilder to reverse the string, as it provides an efficient way to

reverse the word.

2. Step 2: Write a JUnit test class to validate the reverse functionality.

o The test class will use the @Test annotation to mark test methods.

o Use the assertEquals method from JUnit to check if the output matches the

expected reversed string.

3. Step 3: Define sample test cases for various input words, including normal, edge

cases like an empty string, and a single character.

4. Step 4: Execute the tests and validate the output.

Code Implementation:

1. String Reversal Method

The following method reverses a given word using StringBuilder:

java

Copy code

public class StringReversal {

// Method to reverse the input word

public String reverseWord(String word) {

if (word == null) {

return null; // Return null for null input

}

StringBuilder reversed = new StringBuilder(word);

return reversed.reverse().toString(); // Return the reversed string

}

}

2. JUnit Test Class

Here’s the JUnit test class to test the reversal method:

java

Copy code

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

Sample Input:

• Test Case 1:

o Input: "hello"

o Expected Output: "olleh"

• Test Case 2:

o Input: "" (Empty String)

o Expected Output: "" (Empty String)

• Test Case 3:

o Input: "a"

o Expected Output: "a"

• Test Case 4:

o Input: null

o Expected Output: null

Sample Output:

• Test Case 1:

o Input: "hello"

o Output: "olleh"

• Test Case 2:

o Input: "" (Empty String)

o Output: "" (Empty String)

• Test Case 3:

o Input: "a"

o Output: "a"

• Test Case 4:

o Input: null

o Output: null

Results:

The output for each test case should match the expected results:

• Test Case 1: The string "hello" was reversed correctly to "olleh", and the assertion

passed.

• Test Case 2: The empty string "" returned the expected output of "", and the assertion

passed.

• Test Case 3: The single character "a" remained unchanged, as expected, and the

assertion passed.

• Test Case 4: The null input correctly returned null, and the assertion passed.

In conclusion, the JUnit tests confirm that the reverseWord method works correctly for

different input types, including normal words, empty strings, single characters, and null

values.

12.JUnit Test for String Comparison

Aim:

To develop a JUnit test that compares two strings and validates if they are equal using

assertEquals assertions.

Algorithm:

1. Step 1: Create a method to compare two strings.

o The method will take two strings as input and compare them using the equals

method.

2. Step 2: Write a JUnit test class to verify the correctness of the string comparison

functionality.

o The test will use the assertEquals method from JUnit to compare the strings.

3. Step 3: Define test cases with various input combinations, including matching strings,

different strings, and edge cases like empty strings or null values.

4. Step 4: Execute the tests and validate the output against the expected results.

Sample Input:

• Test Case 1:

o Input: "hello", "hello"

o Expected Output: true

• Test Case 2:

o Input: "hello", "world"

o Expected Output: false

• Test Case 3:

o Input: "hello", null

o Expected Output: false

• Test Case 4:

o Input: null, null

o Expected Output: false

• Test Case 5:

o Input: "", "" (Empty strings)

o Expected Output: true

Sample Output:

• Test Case 1:

o Input: "hello", "hello"

o Output: true (The strings are equal)

• Test Case 2:

o Input: "hello", "world"

o Output: false (The strings are different)

• Test Case 3:

o Input: "hello", null

o Output: false (One string is null)

• Test Case 4:

o Input: null, null

o Output: false (Both strings are null)

• Test Case 5:

o Input: "", "" (Empty strings)

o Output: true (Both are empty strings and considered equal)

Results:

The following outcomes are expected from running the test cases:

• Test Case 1: "hello" is equal to "hello", so the test passes.

• Test Case 2: "hello" is not equal to "world", so the test passes.

• Test Case 3: A non-null string is not equal to null, so the test passes.

• Test Case 4: Two null values are not considered equal by the method, so the test

passes.

• Test Case 5: Two empty strings "" are considered equal, so the test passes.

In conclusion, the JUnit tests confirm that the compareStrings method works correctly for

various input scenarios, including equal strings, different strings, null values, and empty .

13.JUnit Test for Voting System (White-Box Testing)

Aim:

To create a JUnit test for a voting system, using white-box testing techniques. This will

involve verifying the system's functionality by testing individual methods and logic paths in

the system.

Algorithm:

1. Step 1: Understand the core functionality of the voting system (e.g., checking

eligibility, casting a vote, and counting votes).

2. Step 2: Design a method for checking voter eligibility (e.g., age greater than or equal

to 18).

3. Step 3: Create a method to cast a vote, where only eligible voters can vote.

4. Step 4: Write a JUnit test class to verify the system's functionality by testing different

scenarios:

o Voter eligibility

o Vote casting

o Vote counting

5. Step 5: Use white-box testing to ensure that each individual logic path (such as valid

or invalid age checks) is thoroughly tested.

Sample Input:

• Test Case 1:

o Input: checkEligibility(18)

o Expected Output: true (Eligible to vote)

• Test Case 2:

o Input: castVote(20, "A")

o Expected Output: true (Vote cast successfully for candidate A)

• Test Case 3:

o Input: castVote(16, "A")

o Expected Output: false (Underage voter, cannot vote)

• Test Case 4:

o Input: castVote(20, "C")

o Expected Output: false (Invalid candidate)

• Test Case 5:

o Input: countVotes("A")

o Expected Output: 2 (Two votes for candidate A)

Sample Output:

• Test Case 1:

o Input: checkEligibility(18)

o Output: true

• Test Case 2:

o Input: castVote(20, "A")

o Output: true

• Test Case 3:

o Input: castVote(16, "A")

o Output: false

• Test Case 4:

o Input: castVote(20, "C")

o Output: false

• Test Case 5:

o Input: countVotes("A")

o Output: 2

Results:

• Test Case 1: The checkEligibility method successfully validated the eligibility of

voters based on age.

• Test Case 2: The castVote method successfully cast votes for eligible voters and

rejected votes for underage voters.

• Test Case 3: The method correctly rejected votes from underage users.

• Test Case 4: The castVote method correctly rejected invalid candidates.

• Test Case 5: The countVotes method correctly counted votes for valid candidates and

returned 0 for invalid candidates.

14.Simple Interest Program with Senior Citizen Rate (White-Box Testing)

Aim:

To write a Java program that calculates simple interest based on different interest rates for

senior citizens (12%) and non-senior citizens (10%). The program will use white-box testing

with JUnit to verify the correctness of the logic.

Algorithm:

1. Step 1: Accept inputs for principal amount, rate of interest, time period, and whether

the customer is a senior citizen.

2. Step 2: Determine the rate of interest:

o If the customer is a senior citizen (age ≥ 60), apply a rate of 12%.

o Otherwise, apply a rate of 10%.

3. Step 3: Calculate simple interest using the formula:

Simple Interest=P×R×T100\text{Simple Interest} = \frac{{P \times R \times

T}}{100}Simple Interest=100P×R×T Where:

o PPP = Principal amount

o RRR = Rate of interest

o TTT = Time period in years

4. Step 4: Write a JUnit test to validate that the correct interest is calculated based on the

provided input.

Sample Input:

• Test Case 1:

o Principal: 1000

o Time: 2 years

o Senior Citizen: true

o Expected Output: 240.0 (12% for senior citizens)

• Test Case 2:

o Principal: 1000

o Time: 2 years

o Senior Citizen: false

o Expected Output: 200.0 (10% for non-senior citizens)

• Test Case 3:

o Principal: 0

o Time: 5 years

o Senior Citizen: true

o Expected Output: 0.0 (Zero principal)

• Test Case 4:

o Principal: 100000

o Time: 10 years

o Senior Citizen: true

o Expected Output: 120000.0 (12% for senior citizens with a large amount)

Sample Output:

• Test Case 1:

o Input: Principal = 1000, Time = 2, Senior Citizen = true

o Output: 240.0

• Test Case 2:

o Input: Principal = 1000, Time = 2, Senior Citizen = false

o Output: 200.0

• Test Case 3:

o Input: Principal = 0, Time = 5, Senior Citizen = true

o Output: 0.0

• Test Case 4:

o Input: Principal = 100000, Time = 10, Senior Citizen = true

o Output: 120000.0

Results:

• Test Case 1: The program correctly calculated the interest for a senior citizen (12%)

with a principal of 1000 and a time of 2 years, resulting in an interest of 240.0.

• Test Case 2: The program correctly calculated the interest for a non-senior citizen

(10%) with the same principal and time, resulting in an interest of 200.0.

• Test Case 3: When the principal is 0, the program correctly returned 0.0 interest,

regardless of the time or senior citizen status.

• Test Case 4: For a large principal (100000) and a long time (10 years), the program

correctly calculated the interest at the senior citizen rate (12%), resulting in 120000.0.

15. Palindrome Check Program (White-Box Testing)

Aim:

To develop a Java program that checks whether a given number is a palindrome and validate

the output using white-box testing with JUnit.

Algorithm:

1. Step 1: Accept the input number (integer).

2. Step 2: Reverse the digits of the number.

3. Step 3: Compare the reversed number with the original number.

4. Step 4: If the original number is equal to the reversed number, it is a palindrome;

otherwise, it is not.

5. Step 5: Write JUnit test cases to verify the correctness of the palindrome logic,

ensuring all possible edge cases are tested.

Sample Input:

• Test Case 1:

o Input: 121

o Expected Output: true (Palindrome)

• Test Case 2:

o Input: 123

o Expected Output: false (Not a palindrome)

• Test Case 3:

o Input: 555

o Expected Output: true (Palindrome)

• Test Case 4:

o Input: 0

o Expected Output: true (Palindrome, edge case)

• Test Case 5:

o Input: -121

o Expected Output: false (Negative number is not a palindrome)

• Test Case 6:

o Input: 1234321

o Expected Output: true (Large palindrome number)

• Test Case 7:

o Input: 987654

o Expected Output: false (Large non-palindrome number)

Sample Output:

• Test Case 1:

o Input: 121

o Output: true

• Test Case 2:

o Input: 123

o Output: false

• Test Case 3:

o Input: 555

o Output: true

• Test Case 4:

o Input: 0

o Output: true

• Test Case 5:

o Input: -121

o Output: false

• Test Case 6:

o Input: 1234321

o Output: true

• Test Case 7:

o Input: 987654

o Output: false

Results:

• Test Case 1: The number 121 is correctly identified as a palindrome.

• Test Case 2: The number 123 is correctly identified as not a palindrome.

• Test Case 3: The number 555 is correctly identified as a palindrome.

• Test Case 4: The number 0 is correctly identified as a palindrome.

• Test Case 5: The negative number -121 is correctly identified as not a palindrome

(edge case).

• Test Case 6: The large number 1234321 is correctly identified as a palindrome.

• Test Case 7: The large number 987654 is correctly identified as not a palindrome.

16.Decimal to Binary and Octal Conversion (White-Box Testing)

Aim:

To write a Java program that converts a decimal number into its binary and octal equivalents

and validate the output using white-box testing with JUnit.

Algorithm:

1. Step 1: Accept the decimal number as input.

2. Step 2: Convert the decimal number into binary using Java's built-in functionality.

3. Step 3: Convert the decimal number into octal using Java's built-in functionality.

4. Step 4: Display the binary and octal equivalents.

5. Step 5: Write JUnit test cases to verify the correctness of the conversion logic,

ensuring that both binary and octal conversions are accurate.

Sample Input:

• Test Case 1:

o Input: 10

o Expected Output:

▪ Binary: 1010

▪ Octal: 12

• Test Case 2:

o Input: 0

o Expected Output:

▪ Binary: 0

▪ Octal: 0

• Test Case 3:

o Input: 255

o Expected Output:

▪ Binary: 11111111

▪ Octal: 377

• Test Case 4:

o Input: -10

o Expected Output:

▪ Binary: -1010

▪ Octal: -12

Sample Output:

• Test Case 1:

o Input: 10

o Output:

▪ Binary: 1010

▪ Octal: 12

• Test Case 2:

o Input: 0

o Output:

▪ Binary: 0

▪ Octal: 0

• Test Case 3:

o Input: 255

o Output:

▪ Binary: 11111111

▪ Octal: 377

• Test Case 4:

o Input: -10

o Output:

▪ Binary: -1010

▪ Octal: -12

esults:

• Test Case 1: The program correctly converts 10 to binary 1010 and octal 12.

• Test Case 2: The program correctly handles the case where the input is 0, outputting

both binary and octal as 0.

• Test Case 3: The program correctly converts 255 to binary 11111111 and octal 377.

• Test Case 4: The program correctly handles the case of a negative number -10,

converting it to binary -1010 and octal -12.

17.Days to Years, Weeks, and Days Conversion (White-Box Testing)

Aim:

To write a Java program that converts a given number of days into years, weeks, and days

and validate the output using white-box testing with JUnit.

Algorithm:

1. Step 1: Accept the number of days as input.

2. Step 2: Calculate the number of years by dividing the total days by 365 (ignoring leap

years).

3. Step 3: Calculate the number of weeks by dividing the remaining days (after

calculating years) by 7.

4. Step 4: Calculate the remaining days after calculating both years and weeks.

5. Step 5: Write JUnit test cases to validate the conversion logic, including edge cases

like 0 days and small numbers of days.

Sample Input:

• Test Case 1:

o Input: 500

o Expected Output:

▪ 1 years, 9 weeks, 1 days

• Test Case 2:

o Input: 1000

o Expected Output:

▪ 2 years, 26 weeks, 6 days

• Test Case 3:

o Input: 1500

o Expected Output:

▪ 4 years, 3 weeks, 5 days

• Test Case 4:

o Input: 0

o Expected Output:

▪ 0 years, 0 weeks, 0 days

• Test Case 5:

o Input: 1

o Expected Output:

▪ 0 years, 0 weeks, 1 days

• Test Case 6:

o Input: 6

o Expected Output:

▪ 0 years, 0 weeks, 6 days

• Test Case 7:

o Input: 365

o Expected Output:

▪ 1 years, 0 weeks, 0 days

Sample Output:

• Test Case 1:

o Input: 500

o Output: 1 years, 9 weeks, 1 days

• Test Case 2:

o Input: 1000

o Output: 2 years, 26 weeks, 6 days

• Test Case 3:

o Input: 1500

o Output: 4 years, 3 weeks, 5 days

• Test Case 4:

o Input: 0

o Output: 0 years, 0 weeks, 0 days

• Test Case 5:

o Input: 1

o Output: 0 years, 0 weeks, 1 days

• Test Case 6:

o Input: 6

o Output: 0 years, 0 weeks, 6 days

• Test Case 7:

o Input: 365

o Output: 1 years, 0 weeks, 0 days

Results:

• Test Case 1: The program correctly converts 500 days into 1 years, 9 weeks, 1 days.

• Test Case 2: The program correctly converts 1000 days into 2 years, 26 weeks, 6

days.

• Test Case 3: The program correctly converts 1500 days into 4 years, 3 weeks, 5 days.

• Test Case 4: The program correctly handles 0 days and returns 0 years, 0 weeks, 0

days.

• Test Case 5: The program correctly handles 1 day and returns 0 years, 0 weeks, 1

days.

• Test Case 6: The program correctly handles 6 days and returns 0 years, 0 weeks, 6

days.

• Test Case 7: The program correctly handles 365 days and returns 1 years, 0 weeks, 0

days.

18.Factorial Calculation Program (White-Box Testing)

Aim:

To write a Java program to calculate the factorial of a number and verify the output using

white-box testing with JUnit.

Algorithm:

1. Step 1: Accept the integer number (n) as input.

2. Step 2: Check if the number is less than 0. If so, return an error message since

factorials are only defined for non-negative integers.

3. Step 3: If the number is 0, return 1, as the factorial of 0 is defined to be 1.

4. Step 4: For any positive integer n, initialize a variable result to 1, and multiply result

by all numbers from 1 to n (inclusive).

5. Step 5: Return the calculated result.

6. Step 6: Write JUnit test cases to validate the factorial calculation logic, including

edge cases like 0, 1, and negative numbers.

Sample Input:

• Test Case 1:

o Input: 5

o Expected Output:

▪ 120 (since 5! = 5 × 4 × 3 × 2 × 1 = 120)

• Test Case 2:

o Input: 3

o Expected Output:

▪ 6 (since 3! = 3 × 2 × 1 = 6)

• Test Case 3:

o Input: 0

o Expected Output:

▪ 1 (since 0! = 1 by definition)

• Test Case 4:

o Input: 1

o Expected Output:

▪ 1 (since 1! = 1)

• Test Case 5:

o Input: -5 (Invalid input)

o Expected Output:

▪ Throws IllegalArgumentException with the message: "Factorial is not

defined for negative numbers."

Sample Output:

• Test Case 1:

o Input: 5

o Output: 120

• Test Case 2:

o Input: 3

o Output: 6

• Test Case 3:

o Input: 0

o Output: 1

• Test Case 4:

o Input: 1

o Output: 1

• Test Case 5:

o Input: -5

o Output: Throws IllegalArgumentException

Results:

• Test Case 1: The program correctly calculates the factorial of 5, outputting 120.

• Test Case 2: The program correctly calculates the factorial of 3, outputting 6.

• Test Case 3: The program correctly handles the edge case of 0, outputting 1.

• Test Case 4: The program correctly calculates the factorial of 1, outputting 1.

• Test Case 5: The program correctly handles negative input, throwing an

IllegalArgumentException as expected.

19.Leap Year Check Program (White-Box Testing)

Aim:

To develop a program that checks whether a given year is a leap year and verify the output

using white-box testing with JUnit.

Algorithm:

1. Step 1: Accept a year as input.

2. Step 2: Check if the year is divisible by 4.

o If the year is divisible by 4, proceed to the next step.

o If not divisible by 4, it is not a leap year.

3. Step 3: Check if the year is divisible by 100.

o If divisible by 100, check if it is divisible by 400.

▪ If divisible by 400, it is a leap year.

▪ If not divisible by 400, it is not a leap year.

4. Step 4: If the year is divisible by 4 but not by 100, it is a leap year.

5. Step 5: Return whether the year is a leap year or not.

6. Step 6: Write JUnit test cases to validate the leap year check, including edge cases

like 0, negative years, and century years.

Sample Input:

• Test Case 1:

o Input: 2020

o Expected Output:

▪ true (2020 is a leap year)

• Test Case 2:

o Input: 2000

o Expected Output:

▪ true (2000 is a leap year)

• Test Case 3:

o Input: 2021

o Expected Output:

▪ false (2021 is not a leap year)

• Test Case 4:

o Input: 1900

o Expected Output:

▪ false (1900 is not a leap year, as it is divisible by 100 but not 400)

• Test Case 5:

o Input: 1600

o Expected Output:

▪ true (1600 is a leap year, divisible by 400)

• Test Case 6:

o Input: -400 (Invalid input)

o Expected Output:-Throws IllegalArgumentException with the message: "Year

cannot be negative."

• Test Case 7:

o Input: 0

o Expected Output:

▪ true (Year 0 is considered a leap year by definition)

Sample Output:

• Test Case 1:

o Input: 2020

o Output: true

• Test Case 2:

o Input: 2000

o Output: true

• Test Case 3:

o Input: 2021

o Output: false

• Test Case 4:

o Input: 1900

o Output: false

• Test Case 5:

o Input: 1600

o Output: true

• Test Case 6:

o Input: -400

o Output: Throws IllegalArgumentException

• Test Case 7:

o Input: 0

o Output: true

Results:

• Test Case 1: The program correctly identifies that 2020 is a leap year.

• Test Case 2: The program correctly identifies that 2000 is a leap year (since it is

divisible by 400).

• Test Case 3: The program correctly identifies that 2021 is not a leap year.

• Test Case 4: The program correctly identifies that 1900 is not a leap year (since it is

divisible by 100 but not by 400).

• Test Case 5: The program correctly identifies that 1600 is a leap year (since it is

divisible by 400).

• Test Case 6: The program correctly throws an IllegalArgumentException for a

negative year input.

• Test Case 7: The program correctly identifies that 0 is a leap year (since 0 is divisible

by 4 and 400).

20.Square and Cube Calculation Program (White-Box Testing)

Aim:

To develop a program that calculates the square and cube of a given decimal number and

verify the output using white-box testing with JUnit.

Algorithm:

1. Step 1: Accept a decimal number as input.

2. Step 2: Calculate the square of the number by multiplying the number by itself.

3. Step 3: Calculate the cube of the number by multiplying the number by itself three

times.

4. Step 4: Return both the square and cube of the given number.

5. Step 5: Write JUnit test cases to validate the square and cube calculation logic,

including edge cases such as 0, 1, and negative numbers.

Sample Input:

• Test Case 1:

o Input: 2.5

o Expected Output:

▪ Square: 6.25

▪ Cube: 15.625

• Test Case 2:

o Input: -3.5

o Expected Output:

▪ Square: 12.25

▪ Cube: -42.875

• Test Case 3:

o Input: 0

o Expected Output:

▪ Square: 0

▪ Cube: 0

• Test Case 4:

o Input: 1

o Expected Output:

▪ Square: 1

▪ Cube: 1

• Test Case 5:

o Input: -1

o Expected Output:

▪ Square: 1

▪ Cube: -1

Sample Output:

• Test Case 1:

o Input: 2.5

o Output:

▪ Square: 6.25

▪ Cube: 15.625

• Test Case 2:

o Input: -3.5

o Output:

▪ Square: 12.25

▪ Cube: -42.875

• Test Case 3:

o Input: 0

o Output:

▪ Square: 0

▪ Cube: 0

• Test Case 4:

o Input: 1

o Output:

▪ Square: 1

▪ Cube: 1

• Test Case 5:

o Input: -1

o Output:

▪ Square: 1

▪ Cube: -1

Results:

• Test Case 1: The program correctly calculates the square (6.25) and cube (15.625) for

2.5.

• Test Case 2: The program correctly calculates the square (12.25) and cube (-42.875)

for -3.5.

• Test Case 3: The program correctly calculates the square (0) and cube (0) for 0.

• Test Case 4: The program correctly calculates the square (1) and cube (1) for 1.

• Test Case 5: The program correctly calculates the square (1) and cube (-1) for -1.

21. Open Google in Chrome using Selenium

Aim:

To write a Selenium script that opens the Google homepage in the Chrome browser.

Prerequisites:

1. Install Selenium WebDriver:

o Install Selenium WebDriver in your project using Maven or Gradle or by

downloading the JAR files from the official Selenium website.

2. Install ChromeDriver:

o Download the appropriate version of ChromeDriver based on your version of

Google Chrome.

o Ensure the chromedriver executable is available in your system's PATH or

specify the location in the script.

Steps to Follow:

1. Step 1: Install required dependencies.

o If using Maven, add the following dependency in your pom.xml file:xml

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.0.0</version> <!-- Make sure to use the latest version -->

</dependency>

2. Step 2: Import necessary classes in your Java code.

3. Step 3: Initialize the ChromeDriver to launch Google in Chrome.

4. Step 4: Launch the Google homepage in Chrome and close the browser after a few

seconds.

Sample Output:

• When you run the script, a Chrome browser window will open.

• The browser will automatically navigate to https://www.google.com.

• After 5 seconds, the browser will close.

Results:

• The Selenium WebDriver successfully opens the Google homepage in the Chrome

browser.

• The browser will be automatically closed after a few seconds.

22. Open Google in Mozilla Firefox using Selenium

Aim:

To write a Selenium script that opens the Google homepage in the Mozilla Firefox browser.

Prerequisites:

1. Install Selenium WebDriver:

o Install Selenium WebDriver in your project using Maven, Gradle, or by

downloading the JAR files from the official Selenium website.

2. Install GeckoDriver:

o Download the appropriate version of GeckoDriver based on your version of

Mozilla Firefox.

o Ensure the geckodriver executable is available in your system's PATH or

specify the location in the script.

Steps to Follow:

1. Step 1: Install the required dependencies.

o If using Maven, add the following dependency in your pom.xml file:

xml

Copy code

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.0.0</version> <!-- Make sure to use the latest version -->

</dependency>

2. Step 2: Import necessary classes in your Java code.

3. Step 3: Initialize the FirefoxDriver to launch Google in Firefox.

4. Step 4: Launch the Google homepage in Firefox and close the browser after a few

seconds.

Sample Input:

• Test Case: Open Google in Firefox

Input: (No user input required, as the script directly opens the URL)

o The URL: https://www.google.com

Sample Output:

• When you run the script:

o A Firefox browser window will open automatically.

o The browser will navigate to https://www.google.com.

o After 5 seconds, the browser will close automatically.

Results:

• The Selenium WebDriver successfully opens the Google homepage in the Mozilla

Firefox browser.

• The browser will automatically close after 5 seconds.

23. Automate ARMS Portal Login using Selenium in Chrome

Aim:

To write a Selenium script to automate the login process to the ARMS Portal using the

Chrome browser.

Prerequisites:

1. Selenium WebDriver:

o Install Selenium WebDriver in your project using Maven or Gradle, or

download the necessary JAR files from the official Selenium website.

2. ChromeDriver:

o Download the appropriate version of ChromeDriver based on your version of

Google Chrome.

o Ensure the chromedriver executable is in your system's PATH or specify the

location of ChromeDriver in the script.

3. ARMS Portal Credentials:

o You should have valid login credentials (username and password) for the

ARMS Portal.

4. Maven Setup (Optional but recommended):

o To install Selenium with Maven, include the following dependency in your

pom.xml:

xml

Copy code

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.0.0</version>

</dependency>

Steps to Follow:

1. Step 1: Install the required dependencies (Selenium and ChromeDriver).

2. Step 2: Import necessary classes in your Java code.

3. Step 3: Initialize ChromeDriver to automate the login process on the ARMS Portal.

4. Step 4: Provide login credentials (username and password) to the portal login page.

5. Step 5: Submit the login form and validate the login.

6. Step 6: Close the browser after a successful login.

Sample Input:

• Username: your\_username (Replace with actual username)

• Password: your\_password (Replace with actual password)

Sample Output:

• When you run the script:

o The Chrome browser will open and navigate to the ARMS Portal login page.

o The script will automatically fill in the username and password and click the

login button.

o After 5 seconds, the browser will close.

o If the login is successful, the portal should load the dashboard or another post-

login page.

Results:

• The Selenium WebDriver successfully automates the login process to the ARMS

Portal in the Chrome browser.

• The username and password are input automatically, and the login form is submitted.

• The browser closes after the login process.

24. Automate GitHub Login using Selenium in Chrome

Aim:

To write a Selenium script to automate the login process to the GitHub portal using the

Chrome browser.

Prerequisites:

1. Selenium WebDriver:

o Install Selenium WebDriver in your project using Maven or Gradle, or

download the necessary JAR files from the official Selenium website.

2. ChromeDriver:

o Download the appropriate version of ChromeDriver based on your version of

Google Chrome.

o Ensure the chromedriver executable is in your system's PATH or specify the

location of ChromeDriver in the script.

3. GitHub Credentials:

o You should have valid login credentials (username and password) for the

GitHub portal.

4. Maven Setup (Optional but recommended):

o To install Selenium with Maven, include the following dependency in your

pom.xml:

xml

Copy code

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.0.0</version>

</dependency>

Steps to Follow:

1. Step 1: Install the required dependencies (Selenium and ChromeDriver).

2. Step 2: Import necessary classes in your Java code.

3. Step 3: Initialize ChromeDriver to automate the login process on the GitHub portal.

4. Step 4: Provide login credentials (username and password) to the portal login page.

5. Step 5: Submit the login form and validate the login.

6. Step 6: Close the browser after a successful login.

Sample Input:

• Username: your\_username (Replace with actual GitHub username)

• Password: your\_password (Replace with actual GitHub password)

Sample Output:

• When you run the script:

o A Chrome browser window will open and navigate to the GitHub login page.

o The script will automatically fill in the username and password fields and

submit the form.

o After 5 seconds, the browser will close.

o If the login is successful, the script will load the GitHub dashboard or home

page.

Results:

• The Selenium WebDriver successfully automates the login process to GitHub using

the Chrome browser.

• The username and password are input automatically, and the login form is submitted.

• The browser will close after the script finishes executing.

25. Automate Swiggy Login using Selenium in Chrome

Aim:

To write a Selenium script to automate the login process to the Swiggy portal using the

Chrome browser.

Prerequisites:

1. Selenium WebDriver:

o Install Selenium WebDriver in your project using Maven or Gradle, or

download the necessary JAR files from the official Selenium website.

2. ChromeDriver:

o Download the appropriate version of ChromeDriver based on your version of

Google Chrome.

o Ensure that chromedriver is in your system's PATH or specify the location of

ChromeDriver in the script.

3. Swiggy Credentials:

o You should have valid login credentials (username and password) for the

Swiggy portal.

4. Maven Setup (Optional but recommended):

o To install Selenium with Maven, include the following dependency in your

pom.xml:

xml

Copy code

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.0.0</version>

</dependency>

Steps to Follow:

1. Step 1: Install the required dependencies (Selenium and ChromeDriver).

2. Step 2: Import necessary classes in your Java code.

3. Step 3: Initialize ChromeDriver to automate the login process on the Swiggy portal.

4. Step 4: Provide login credentials (username and password) to the Swiggy login page.

5. Step 5: Submit the login form and validate the login.

6. Step 6: Close the browser after a successful login.

o These methods simulate typing the phone number/email and OTP into the

respective fields.

2. loginButton.click();

o This simulates clicking the login button to submit the login form.

3. Thread.sleep(5000);

Sample Input:

• Phone/Email: your\_phone\_or\_email (Replace with actual phone number or email

used for Swiggy login)

• OTP: your\_otp (Replace with the OTP received on your phone or email)

Sample Output:

• When you run the script:

o A Chrome browser window will open and navigate to the Swiggy homepage.

o The script will automatically click the login button, enter the phone number or

email, and submit the OTP.

o After successful login, the home page will be loaded.

o The browser will close after a 5-second wait.

Results:

• The Selenium WebDriver successfully automates the login process to Swiggy using

the Chrome browser.

• The phone number/email and OTP are input automatically, and the login form is

submitted.

• The browser will close after the script finishes executing.

26. Matrix Multiplication Program in C

Aim:

To write a C program to multiply two matrices and document the causes if the program fails

for certain dimensions (e.g., dimension mismatch, uninitialized values).

Prerequisites:

1. Basic knowledge of matrix multiplication.

2. Basic understanding of 2D arrays in C.

3. A C compiler installed (e.g., GCC, Turbo C, etc.).

Algorithm:

1. Input the Dimensions:

o Read the number of rows and columns for both matrices (Matrix A and Matrix

B).

2. Matrix Multiplication Validity Check:

o Ensure that the number of columns in Matrix A is equal to the number of rows

in Matrix B. If this condition is not met, print an error message and exit the

program.

3. Input Matrix Elements:

o Read the elements of Matrix A and Matrix B.

4. Matrix Multiplication Logic:

o Initialize the result matrix with dimensions of rowsA x columnsB.

o Multiply the corresponding rows of Matrix A and columns of Matrix B.

o Store the result in the result matrix.

5. Output the Result Matrix:

o Print the resulting matrix.

Sample Input:

Enter the number of rows and columns for Matrix A: 2 3

Enter the number of rows and columns for Matrix B: 3 2

Enter the elements of Matrix A:

1 2 3

4 5 6

Enter the elements of Matrix B:

7 8

9 10

11 12

Sample Output:

Resultant Matrix after multiplication:

58 64

139 154

Common Errors and Causes:

1. Dimension Mismatch:

o If the number of columns in Matrix A does not match the number of rows in

Matrix B (colsA != rowsB), the program will print an error message and exit.

This is a common issue when performing matrix multiplication, as it is a

fundamental requirement.

2. Uninitialized Values:

o If the elements of the matrices are not properly initialized, the program might

use garbage values, leading to incorrect results. Ensure that all matrix

elements are correctly entered before proceeding with the multiplication.

27. Bubble Sort with Index Out of Bounds Error Simulation and Prevention

Aim:

To implement a bubble sort algorithm in C, simulate an "index out of bounds" error, analyze

the causes, and propose precautions to prevent such errors.

Algorithm:

1. Input the dimensions of the array:

o Read the number of elements n.

2. Input the elements of the array:

o Read the elements of the array into an array arr[].

3. Simulate index out of bounds error:

o Loop through the array and attempt to access an element at an invalid index

(e.g., arr[n]).

4. Bubble Sort Logic:

o Iterate over the array, compare adjacent elements, and swap them if they are in

the wrong order.

5. Error Handling:

o If an "index out of bounds" error occurs, print an error message and exit.

Sample Input:

Enter the number of elements: 5

Enter the elements of the array:

12 45 23 4 56

Sample Output:

Attempting to access arr[5], which is out of bounds!

Results:

• Error Simulation: The program attempts to access an array index (arr[5]) that is out

of bounds for an array of size 5. This results in an "index out of bounds" error.

• Precaution: Ensure that loops and array accesses are within valid index ranges (0 to

n-1) to prevent this error .

28. File Operations in C with Error Handling

Aim:

To write a C program to perform file operations such as reading from and writing to a file,

and test for failure scenarios such as missing files, permission issues, or incorrect paths.

Document the findings and suggest fixes.

Algorithm:

1. Opening a File:

o Use fopen() to open a file in "read" or "write" mode.

o Check whether the file was successfully opened by verifying the returned file

pointer.

2. Reading and Writing to a File:

o Use fscanf() or fgets() to read from a file.

o Use fprintf() or fputs() to write to a file.

3. Failure Scenarios:

o Test failure scenarios by providing incorrect file paths, missing files, or files

with insufficient permissions.

o Handle errors using fopen() and check for NULL return values.

o Print appropriate error messages using perror() or custom error messages.

4. Closing the File:

o Always close the file after operations using fclose() to free resources.

Code Implementation:

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*file;

char filename[100];

char content[] = "Hello, this is a test file.\n";

// Step 1: Ask user for the file name

printf("Enter the file name to read from and write to: ");

scanf("%s", filename);

// Step 2: Open the file for writing (create if doesn't exist)

file = fopen(filename, "w");

if (file == NULL) {

perror("Error opening file for writing");

return 1; // Exit the program if unable to open the file

}

// Step 3: Write to the file

fprintf(file, "%s", content);

printf("Content written to file: %s\n", content);

// Step 4: Close the file after writing

fclose(file);

// Step 5: Reopen the file for reading

file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file for reading");

return 1; // Exit the program if unable to open the file for reading

}

// Step 6: Read from the file

char buffer[255];

while (fgets(buffer, sizeof(buffer), file)) {

printf("Read from file: %s", buffer);

}

// Step 7: Close the file after reading

fclose(file);

return 0;

}

Explanation of the Code:

1. File Opening:

o The program opens the file in "w" mode for writing. If the file does not exist,

it is created. If the program cannot open the file, it prints an error using

perror().

2. Writing to the File:

o The program writes a string "Hello, this is a test file.\n" to the file using

fprintf().

3. Reading from the File:

o After closing the file, the program reopens it in "r" mode for reading. If the

file cannot be opened, an error message is displayed using perror().

o The program reads the content of the file using fgets() and prints it to the

console.

4. Error Handling:

o The program handles errors using perror() when file operations fail (e.g., due

to missing files, incorrect paths, or insufficient permissions).

5. File Closing:

o After each file operation (reading or writing), the file is closed using fclose()

to free resources.

Sample Input:

Enter the file name to read from and write to: testfile.txt

Sample Output (if the file exists and operations are successful):

Failure Scenarios:

1. Missing File:

o If the specified file does not exist when opening in read mode

(fopen(filename, "r")), the program will print an error message like:

yaml

Copy code

Error opening file for reading: No such file or directory

2. Permission Issues:

o If the file is not writable due to permission issues, the program will print:

javascript

Copy code

Error opening file for writing: Permission denied

3. Incorrect File Path:

o If the path to the file is incorrect or the directory doesn't exist, the program

will print:

yaml

Copy code

Error opening file for writing: No such file or directory

4. Handling Errors Gracefully:

o The program uses perror() to print specific error messages based on the failure

scenario, providing useful information for troubleshooting.

Suggestions for Fixes:

1. Ensure Correct File Path:

o Always provide the correct path, especially if the file is located in a different

directory. Use absolute paths if necessary.

2. Check Permissions:

o Ensure that the file has appropriate read and write permissions. You can

change file permissions using commands like chmod in Linux.

3. Verify File Existence:

o Before reading a file, ensure that the file exists. You can use access() to check

the existence of a file.

4. Use File Path Validation:

o Ensure the directory exists where the file is being created or read from. You

can use mkdir() to create directories if needed.

Results:

This program demonstrates file operations in C, including reading from and writing to files. It

handles errors that may occur due to missing files, permission issues, or incorrect file paths.

By using proper error handling and ensuring correct file paths and permissions, file

operations can be performed successfully in C programs.

29.Recursive Factorial Function in C

Aim:

To implement a recursive factorial function in C and analyze its behavior when the recursion

depth exceeds the stack size. Propose solutions to handle this issue gracefully.

Algorithm:

1. Base Case:

o If the input number is 0 or 1, return 1 as the factorial of 0 and 1 is always 1.

2. Recursive Case:

o If the input number n is greater than 1, return n \* factorial(n-1).

3. Handle Stack Overflow:

o As recursion depth increases, the program consumes more stack space. When

the recursion depth exceeds the available stack size, a stack overflow may

occur.

o Propose ways to limit recursion depth or convert the recursive function to an

iterative version to avoid stack overflow.

Sample Input:

Enter a positive integer: 5

Sample Output:

Factorial of 5 is: 120

Sample Input for Iterative Approach:

Enter a positive integer: 5

Sample Output for Iterative Approach:

Factorial of 5 is: 120

Results:

• Recursive Approach:

o For small numbers (e.g., 5), the recursive approach works correctly and

provides the correct result.

o For large numbers, the recursion depth may exceed the available stack size,

causing a stack overflow unless the recursion depth is limited.

• Iterative Approach:

o The iterative approach avoids recursion and does not risk stack overflow,

making it more suitable for calculating the factorial of large numbers.

30.Dynamic Memory Allocation in C and Detecting Memory Leaks

Aim:

To write a C program that dynamically allocates memory for an array and uses tools like

Valgrind to detect memory leaks. Document the findings and propose solutions to prevent

memory leaks.

Algorithm:

1. Dynamic Memory Allocation:

o Use malloc() or calloc() to dynamically allocate memory for an array based on

user input.

o Check if memory allocation is successful by verifying that the returned pointer

is not NULL.

2. Array Operations:

o Populate the dynamically allocated array with user input or predefined values.

o Perform operations on the array (e.g., print values, sum elements).

3. Memory Deallocation:

o Use free() to deallocate the dynamically allocated memory after it is no longer

needed.

4. Memory Leak Detection:

o Use tools like Valgrind to check for memory leaks and ensure that memory is

properly freed after use.

5. Error Handling:

o If memory allocation fails, print an appropriate error message and exit the

program.

Sample Input:

Enter the number of elements: 5

Enter the elements of the array:

1 2 3 4 5

Sample Output:

The elements of the array are:

1 2 3 4 5

Memory freed successfully.

31.Power Calculation with Overflow Detection in C

Aim:

To write a C program that calculates the power of a number (e.g., base^exponent) and detects

scenarios where integer overflow occurs. The program will identify the conditions leading to

overflow and suggest strategies to avoid it.

Algorithm:

1. Input:

o Take two integers as input: base (the base number) and exponent (the power to

which the base is raised).

2. Power Calculation:

o Initialize a result variable to 1.

o Multiply the result by the base for the number of times specified by the

exponent.

3. Overflow Detection:

o During the power calculation, check if multiplying the result by the base

would exceed the maximum value that can be stored in the integer type

(INT\_MAX for a 32-bit system).

o If overflow is detected, print an error message and stop further computation.

4. Error Handling:

o Handle negative exponents (e.g., base^-n is 1 / base^n).

o Provide feedback when the result is too large to fit in the data type's limits.

5. Strategies to Avoid Overflow:

o Use larger data types like long long or double for large numbers.

o Implement checks during multiplication to detect potential overflow before it

happens.

Sample Input:

Enter base: 10

Enter exponent: 10

Sample Output:

10^10 = 10000000000

32.Sum of Digits to Single Digit with Infinite Loop Simulation

Aim:

To create a program that repeatedly calculates the sum of digits of a number until a single

digit is obtained. The program will simulate an infinite loop condition, investigate its cause,

and provide strategies to prevent and resolve such loops.

Algorithm:

1. Input:

o Take an integer input from the user.

2. Sum of Digits Calculation:

o Use a loop to extract and sum the digits of the number.

o After calculating the sum, check if the sum is a single digit (i.e., less than 10).

o If not, repeat the process by setting the sum as the new number.

3. Infinite Loop Simulation:

o Intentionally introduce a condition where the program will get stuck in an

infinite loop (e.g., if the user inputs a non-positive number or an inappropriate

value).

4. Detecting the Infinite Loop:

o Investigate and identify the cause of the infinite loop.

o Provide methods to resolve this, such as adding checks for invalid inputs.

5. Loop Prevention Strategy:

o Ensure the input is valid (positive integer) to prevent unintended infinite loops.

Sample Input:

Enter a number: 9876

Sample Output:

Current sum of digits: 30

Current sum of digits: 3

Single digit obtained: 3

Sample Input (Invalid Input):

Enter a number: -123

Sample Output (Error Message):

Invalid input! Please enter a positive integer.

Results:

Thus the program that repeatedly calculates the sum of digits of a number until a single digit

is obtained.

33.Producer-Consumer Problem with Deadlock Simulation

Aim:

To write a multi-threaded C program simulating the producer-consumer problem. The

program will intentionally introduce a deadlock scenario, analyze its causes, and suggest

techniques like using mutexes or semaphores to resolve the deadlock.

Algorithm:

1. Initialize the Buffer:

o The buffer is used for storing items produced by the producer and consumed

by the consumer.

o The buffer should have a fixed size, and we will use two pointers or indices to

keep track of the start and end positions.

2. Producer Thread:

o The producer will generate items and put them in the buffer.

o The producer will check if the buffer is full before adding new items. If full,

the producer will wait.

3. Consumer Thread:

o The consumer will consume items from the buffer.

o The consumer will check if the buffer is empty before consuming. If empty,

the consumer will wait.

4. Deadlock Simulation:

o Introduce a condition where both the producer and the consumer are stuck

waiting for each other, causing a deadlock.

5. Mutexes/Semaphores for Synchronization:

o Use mutexes or semaphores to synchronize access to shared resources (the

buffer).

o Ensure that the producer and consumer threads do not access the buffer

simultaneously to prevent race conditions.

6. Deadlock Resolution:

o After identifying the causes of the deadlock, implement proper

synchronization using mutexes and semaphores to resolve the deadlock.

o The mutex is used to ensure mutual exclusion when accessing the shared

buffer.

Results:

The deadlock in the producer-consumer problem was simulated and resolved using

semaphores. By introducing semaphores to handle synchronization between the producer and

consumer threads, we resolved the deadlock and ensured smooth operation of the system. The

program is now free from deadlocks and is capable of efficiently managing buffer space

between producing and consuming items.

34.Basic Calculator with Division by Zero Handling

Aim:

To develop a basic calculator program in C that performs arithmetic operations (addition,

subtraction, multiplication, and division). The program will handle division by zero scenarios

by implementing appropriate error-handling mechanisms.

Algorithm:

1. Input Handling:

o The program will prompt the user to enter two numbers and select an

arithmetic operation (addition, subtraction, multiplication, or division).

2. Operation Selection:

o Based on the user’s choice, the program will perform the corresponding

arithmetic operation.

3. Division by Zero Check:

o Before performing division, the program will check if the divisor is zero.

o If the divisor is zero, the program will display an error message and prompt

the user again to input valid numbers.

4. Error Handling:

o For invalid input (non-numeric values or zero in the denominator), the

program will display an error message and ask the user to re-enter the input.

5. Display Result:

o After performing the selected operation, the program will display the result.

6. Repeat or Exit:

o The user can choose to repeat the operations or exit the program.

Sample Input:

Basic Calculator:

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 4

Enter two numbers: 10 0

Sample Output:

Basic Calculator:

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 4

Enter two numbers: 10 0

Error: Division by zero is not allowed.

Sample Input (Valid Division):

Basic Calculator:

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 4

Enter two numbers: 10 2

Sample Output (Valid Division):

Basic Calculator:

1. Add

2. Subtract

3. Multiply

4. Divide

5. Exit

Enter your choice (1-5): 4

Enter two numbers: 10 2

Result: 5.00

Results:

The program successfully handles division by zero by checking the denominator before

attempting division and displaying an error message when necessary. Error handling for

invalid choices and invalid input ensures the program runs smoothly without crashes or

undefined behavior.

35. Quicksort with Performance Analysis

Aim:

To implement the quicksort algorithm in C and simulate a scenario where the algorithm

performs poorly due to selecting a poor pivot. The goal is to analyze the causes of

performance degradation and propose optimizations to improve efficiency.

Algorithm:

1. Input Handling:

o Input an array of integers to be sorted using the quicksort algorithm.

2. Quicksort Implementation:

o Select a pivot element from the array (this example uses the first element as

the pivot).

o Partition the array into two sub-arrays: one with elements smaller than the

pivot and the other with elements larger than the pivot.

o Recursively apply the same process to the sub-arrays.

3. Performance Analysis:

o Measure the time taken for sorting using the poor pivot strategy (e.g.,

choosing the first element as pivot).

o Simulate a scenario where the array is nearly sorted or reversed, which leads

to poor performance (O(n2)).

4. Proposed Optimizations:

o Implement a better pivot selection method, such as choosing the median

element or using randomized pivoting, to improve performance (reduce the

likelihood of O(n2) complexity).

Sample Input:

Enter the number of elements: 10

Enter the elements: 10 9 8 7 6 5 4 3 2 1

Sample Output (Poor Performance):

Sorted array: 1 2 3 4 5 6 7 8 9 10

Time taken for sorting: 0.000004 seconds

Results:

• The quicksort algorithm performs poorly when a bad pivot is chosen (e.g., the first

element).

• The performance can degrade to O(n2) in the worst case, especially for sorted or

nearly sorted data.

36. Peer Review for Test Cases and Test Plans

Aim:

To implement and practice a peer review process for test cases and test plans to improve their

quality and identify potential issues early in the testing process. The goal is to ensure that the

test cases are comprehensive, accurate, and aligned with project requirements.

Algorithm:

1. Develop Test Plan and Test Cases:

o First, develop a detailed test plan that outlines the objectives, scope, resources,

schedule, and criteria for success.

o Develop individual test cases based on the requirements of the system,

considering positive and negative scenarios.

2. Peer Review Process:

o Preparation: The test plan and test cases are shared with peers (team

members or other stakeholders) for review.

o Review: Each reviewer examines the test cases for:

▪ Completeness: Ensuring all functionalities of the system are covered.

▪ Accuracy: Verifying that test cases correctly implement the intended

behavior.

▪ Correctness: Checking if the expected results are aligned with the

actual system behavior.

▪ Clarity: Ensuring that the test cases are clear, concise, and easy to

understand.

▪ Edge Cases: Identifying potential boundary or edge cases not

considered in the original test cases.

o Feedback: Reviewers provide feedback, suggestions, and identify possible

errors or gaps in the test cases or plan.

3. Addressing Feedback:

o The tester who created the test cases revises the test plan and test cases based

on the feedback received, improving the quality of the documentation.

4. Final Review and Approval:

o Once revisions are made, the test plan and test cases go through a final review

to ensure all issues are addressed.

o The test plan and cases are then approved and ready for execution.

Steps to Implement Peer Review:

1. Step 1: Test Plan Creation

o Create a detailed test plan that includes:

▪ Test Plan ID: Unique identifier.

▪ Test Items: List of software components being tested.

▪ Test Objectives: The goals for testing.

▪ Test Scope: What features are in and out of scope for testing.

▪ Resources: Hardware and software resources required.

▪ Schedule: Timeframes and deadlines.

▪ Criteria for Success: Conditions that must be met for the test to be

considered successful.

2. Step 2: Test Case Development

o Create test cases based on the test plan. Ensure they cover:

▪ Test Case ID: Unique identifier.

▪ Description: A short explanation of the test case.

▪ Test Steps: A detailed list of actions to perform.

▪ Expected Result: The anticipated output or behavior.

▪ Actual Result: Space for noting the real outcome during testing.

▪ Pass/Fail Criteria: Define the conditions under which the test will

pass or fail.

3. Step 3: Peer Review

o Share the test plan and test cases with a peer reviewer for review.

o Reviewers should focus on completeness, accuracy, clarity, and identifying

any edge cases that may have been overlooked.

o Use a collaborative tool (e.g., Google Docs, JIRA, etc.) for feedback

collection.

4. Step 4: Incorporate Feedback

o Review the feedback carefully, make necessary changes to the test plan and

test cases, and improve the quality based on peer comments.

5. Step 5: Final Approval

o Submit the revised test cases and test plan for final approval to the project

manager or lead tester.

o Once approved, the test plan and test cases are ready for execution.

Sample Test Plan:

Test Plan ID: TP001

Test Items: Login functionality of the Web Application

Test Objectives: Ensure the login page accepts valid credentials and rejects invalid

credentials.

Test Scope:

• In-scope: Username, password fields, login button, remember me functionality.

• Out-of-scope: UI components like images and design elements.

Resources Required:

• Web browser (Chrome or Firefox)

• User credentials (valid/invalid)

Schedule:

• Review of test plan: 1 day

• Test execution: 3 days

• Report generation: 1 day

Success Criteria:

• Login successful with valid credentials.

• Appropriate error message displayed for invalid credentials.

Sample Test Case:

Test Case ID: TC001

Test Case Name: Valid Login Test

Description: Test that the system allows users to log in with valid credentials.

Test Steps:

1. Navigate to the login page.

2. Enter a valid username and password.

3. Click the login button.

4. Verify that the user is redirected to the homepage.

Expected Result:

The user should be successfully redirected to the homepage after clicking the login button.

Actual Result:

(To be filled during test execution.)

Pass/Fail Criteria:

Pass if the user is redirected to the homepage; Fail if an error occurs.

Sample Feedback from Peer Review:

1. Completeness:

o Add edge cases for password length (e.g., empty password or password

exceeding max length).

2. Clarity:

o Specify that the username and password should be valid in the form of a

database entry (not just a random string).

3. Accuracy:

o Ensure that the test step clearly defines which login button to click (e.g., main

login button vs. submit button).

4. Edge Cases:

o Consider cases where the user leaves the username or password fields empty

and test the behavior.

Sample Revised Test Case:

Test Case ID: TC001

Test Case Name: Valid Login Test

Description: Test that the system allows users to log in with valid credentials.

Test Steps:

1. Navigate to the login page.

2. Enter a valid username (e.g., "testuser") and a valid password (e.g., "Test1234").

3. Click the login button.

4. Verify that the user is redirected to the homepage.

Expected Result:

The user should be successfully redirected to the homepage after clicking the login button.

Actual Result:

(To be filled during test execution.)

Pass/Fail Criteria:

Pass if the user is redirected to the homepage; Fail if an error occurs.

Edge Case Test:

• Test the login behavior with empty username or password fields.

Sample Performance Metrics for Review:

• Review Duration:

o Test Plan Review: 1 day

o Test Case Review: 2 days

• Number of Issues Identified:

o 5 issues identified during peer review (e.g., missing edge cases, unclear test

steps).

• Time to Resolve Feedback:

o Feedback resolution: 1 day

Conclusion:

The peer review process is an essential part of ensuring the quality and thoroughness of test

cases and test plans. By using structured peer reviews, teams can identify gaps, improve

clarity, and prevent overlooking critical test scenarios. This process enhances the overall

quality of the testing efforts and ensures that the final product meets user requirements

effectively.

37. Defect Tracking with Tools (Jira/Bugzilla)

Aim:

To use defect tracking tools such as Jira or Bugzilla for reporting testing progress, tracking

defect resolution, and managing the testing lifecycle effectively.

Algorithm:

1. Set Up Defect Tracking Tool (Jira/Bugzilla):

o Install and configure the defect tracking tool (Jira or Bugzilla).

o Set up the necessary project, users, and access roles to ensure that everyone

involved in the project can access and update defect information.

2. Defect Reporting Process:

o Identify Defects: During testing, whenever a defect (bug) is found, it is

reported in the defect tracking tool.

o Create a Defect Ticket: The tester or developer creates a defect ticket in the

tool.

▪ Include details such as:

▪ Summary: A short description of the defect.

▪ Description: A detailed description, including steps to

reproduce, expected result, and actual result.

▪ Priority: The urgency level (e.g., Low, Medium, High).

▪ Severity: The impact level (e.g., Critical, Major, Minor).

▪ Assignee: The person responsible for fixing the defect.

▪ Status: The current status of the defect (e.g., Open, In Progress,

Resolved).

▪ Attachments: Any relevant files, screenshots, or logs that help

in understanding the defect.

o Link Defects to Test Cases: Ensure that the defect is linked to the

corresponding test case or test plan.

3. Defect Tracking Process:

o Monitor the Defects: Regularly monitor the defect tickets for updates and

resolutions.

o Defect Status Updates: As defects are worked on, the status should be

updated (e.g., from Open to In Progress, and finally Resolved).

o Re-Testing: Once defects are resolved, the tester re-tests the defect to confirm

the fix.

o Defect Closure: When the defect is verified and resolved, it is closed.

4. Generate Defect Reports:

o Use built-in reporting features in Jira or Bugzilla to track defect resolution

progress.

▪ Reports can include:

▪ Total defects reported.

▪ Defects closed vs. open.

▪ Defect severity distribution.

▪ Average time to resolve defects.

5. Communicate Defect Status to Stakeholders:

o Share the progress with project managers and stakeholders using defect reports

or dashboards.

Steps for Using Jira or Bugzilla:

Using Jira for Defect Tracking:

1. Create a Project in Jira:

o Navigate to Jira Dashboard → Projects → Create Project.

o Select the project type (e.g., Software, Business, etc.) and configure it.

2. Create a Defect (Bug) Ticket:

o Go to Backlog or Issues.

o Click on Create Issue.

o Choose Bug as the issue type.

o Fill in necessary details (Summary, Description, Priority, etc.).

o Click Create.

3. Update the Defect Ticket:

o To update a defect, click on the ticket.

o Change its Status (e.g., "In Progress", "Resolved").

o Add Comments to describe the progress or any issues encountered.

o Attach files (e.g., screenshots, logs) if necessary.

4. Re-test Defects:

o After a defect is fixed, verify the fix by re-testing.

o If the defect is resolved, mark it as Closed.

o If the defect persists, update the ticket with more information or reassign it for

further investigation.

5. Generate Defect Reports:

o Go to Reports in Jira and choose a defect tracking report (e.g., Defects by

Priority, Defects Resolution Time).

Using Bugzilla for Defect Tracking:

1. Create a Product in Bugzilla:

o Login to Bugzilla and navigate to Admin Panel → Create Product.

o Define product details and project configuration.

2. Create a Defect (Bug) Ticket:

o Go to Enter Bug.

o Select the relevant Product and Component.

o Fill in necessary details (Summary, Description, Priority, etc.).

o Click Submit Bug.

3. Update the Defect Ticket:

o Click on the defect ticket.

o Update its Status (e.g., “In Progress”, “Resolved”).

o Add Comments and Attachments as required.

4. Re-test Defects:

o Once the developer resolves the defect, re-test it.

o If it’s fixed, update the status to Resolved or Closed.

o If not, add more details and reassign it for further analysis.

5. Generate Defect Reports:

o Go to Reports → Bug Summary and generate reports (e.g., total open bugs,

by severity).

Sample Defect Report in Jira:

Defect

ID

Summary Priority Severity Status Assignee Created

Date

Resolved

Date

DEF001 Login Button

Not Working

Page Not

High Critical Resolved John Doe 01/15/2024 01/16/2024

In

DEF002 Loading

Properly

Medium Major Progress

Jane Smith 01/16/2024 -

DEF003 Form Field

Validation Error

Low Minor Closed Mike

Johnson 01/14/2024 01/15/2024

Sample Defect Report in Bugzilla:

Bug ID Summary Priority Severity Status Assigned

To

Created

On

Resolved

On

BUG001 Login Form Error High Critical Resolved Sarah Lee 02/01/2024 02/02/2024

BUG002 Database

Connection Issue

Medium Major In

Progress

Tom

Green 02/02/2024 -

BUG003 UI Misalignment Low Minor Closed Amy

White 01/30/2024 01/31/2024

Results:

Using defect tracking tools like Jira and Bugzilla significantly enhances the defect

management process. These tools provide visibility into the progress of defect resolution,

help maintain detailed records of defect history, and streamline communication between

testers, developers, and stakeholders. By tracking defects efficiently, the software

development and testing teams can ensure faster issue resolution and higher-quality products.

38.Implement Version Control and Configuration Management for Test Scripts and

Environments

Aim:

To implement version control and configuration management for test scripts and test

environments to ensure efficient management, collaboration, and reproducibility in the testing

process.

Algorithm:

1. Set Up a Version Control System (VCS):

o Choose a version control system (e.g., Git) to manage test scripts and

configuration files.

o Initialize a Git repository for the project (if not already done).

▪ Command: git init

o Create a .gitignore file to exclude unnecessary files (e.g., temporary files,

logs).

o Add all relevant test scripts, configuration files, and environment setups to the

Git repository.

▪ Command: git add .

o Commit the changes with meaningful commit messages.

▪ Command: git commit -m "Initial commit of test scripts and

configurations"

2. Branch Management:

o Create separate branches for different types of work, such as feature branches

for new tests or bug fix branches.

▪ Command: git branch <branch\_name>

▪ Command: git checkout <branch\_name>

o Merge the feature branches into the main branch after testing.

▪ Command: git merge <branch\_name>

o Resolve any merge conflicts that arise.

3. Versioning Test Scripts:

o Maintain versioned commits of test scripts to track changes over time.

o Tag specific commits with version numbers to indicate releases or major

changes.

▪ Command: git tag v1.0.0

o Use Git diff to review changes between different versions of test scripts.

▪ Command: git diff <commit1> <commit2>

4. Configuration Management for Test Environments:

o Store environment configuration files (e.g., environment variables, config

files) under version control.

o Use configuration management tools (e.g., Ansible, Puppet, or Chef) to

automate the setup and teardown of test environments.

o Maintain a README.md or Environment Setup Guide to document the

environment configuration process.

o If using Docker, store Dockerfile and Docker Compose files to define the

testing environment.

5. Automating Test Environment Deployment:

o Use tools like Docker or Vagrant to automate the creation of consistent testing

environments across different machines.

o For Docker:

▪ Create a Dockerfile to define the testing environment.

▪ Create a docker-compose.yml file to set up multiple services (e.g.,

databases, web servers).

o Use environment variables to customize the environment for different test

scenarios.

▪ Command: docker-compose up --build

o Verify that the environment setup is consistent across all systems.

6. Continuous Integration (CI) Integration:

o Integrate version control and test environment setup with a CI/CD pipeline

(e.g., Jenkins, GitHub Actions, GitLab CI).

o Configure the pipeline to automatically pull the latest test scripts from the

repository and run the tests on the configured environment.

o Ensure that every change to the test scripts or environment is tracked, tested,

and reported through CI tools.

7. Monitoring and Reporting:

o Implement reporting tools (e.g., Allure, TestNG, or JUnit reports) to track test

execution and results.

o Store test results and logs in the version control repository or in a dedicated

artifacts repository.

Results:

Implementing version control and configuration management for test scripts and

environments ensures consistency, collaboration, and effective defect tracking. Integrating

with CI/CD pipelines further improves efficiency and provides faster feedback to developers

and testers.

39. Use Git, SVN, or Mercurial for Managing Test Scripts, Defect Logs, and Test Results

Aim:

To utilize version control tools like Git, SVN, or Mercurial to efficiently manage test scripts,

defect logs, and test results, ensuring easy tracking of changes, collaboration, and

reproducibility across different environments.

Algorithm:

1. Choose a Version Control Tool:

o Select one of the tools (Git, SVN, or Mercurial) to manage your project files.

o Git: A distributed version control system suitable for collaboration and

branching.

o SVN (Subversion): A centralized version control system, good for single

repositories.

o Mercurial: Another distributed version control system, similar to Git but

simpler to use in some cases.

2. Initialize a Repository:

o For Git: Initialize a Git repository in the project folder.

git init

o For SVN: Create a repository on a server or local machine and check out the

directory.

svn checkout <repository\_url>

o For Mercurial: Initialize a new repository in the project directory.

hg init

3. Add Files to Version Control:

o For Git:

git add test-scripts/\*

git add defect-logs/\*

git add test-results/\*

o For SVN: Add files to the repository with the svn add command.

svn add test-scripts/\* defect-logs/\* test-results/\*

o For Mercurial: Add files to the repository with the hg add command.

hg add test-results/\*

hg commit -m "Added test results for regression testing"

7. Collaborate with Branches (Optional):

o For Git:

▪ Create a new branch for testing or feature development.

git branch feature/new-test-case

git checkout feature/new-test-case

▪ After completing the work, merge it into the main branch:

git checkout main

git merge feature/new-test-case

o For SVN: Create a new branch using svn copy:

svn copy trunk branches/feature/new-test-case

o For Mercurial: Create a new branch using hg branch:

hg branch feature/new-test-case

hg commit -m "Created new feature branch for test case"

8. Tagging Releases (Optional):

o Tag important versions or milestones (e.g., major test script updates).

o For Git:git tag v1.0.0

git push --tags

o For SVN:svn copy trunk tags/v1.0.0

svn commit -m "Tagging version 1.0.0"

o For Mercurial:

Results:

hg tag v1.0.0

hg push

Using version control systems like Git, SVN, or Mercurial to manage test scripts, defect logs,

and test results offers several benefits, including improved collaboration, traceability, and

reproducibility of the testing process. By integrating version control, teams can ensure that

their testing artifacts are organized, easily accessible, and consistent across different testing

environment.

40. Perform Test Closure Activities, Including Documentation, Analysis, and Final

Reporting for a Project

Aim:

To conduct test closure activities, ensuring proper documentation, analysis of testing

outcomes, and final reporting to provide a comprehensive review of the testing process and

results, ensuring that all necessary tasks have been completed and the project is ready for

closure.

Algorithm:

1. Review Test Completion Criteria:

o Confirm that all planned test cases have been executed.

o Ensure that the testing objectives have been met (e.g., number of defects

found, severity of issues, etc.).

o Verify that no critical test cases or scenarios have been overlooked.

2. Document Test Results:

o Record the results of all executed test cases (pass/fail).

o Log the number of defects identified, their severity, and the status of each

defect (open, resolved, closed).

o Document any deviations from the initial test plan or schedule, and their

reasons.

3. Analyze Defects and Test Coverage:

o Perform defect analysis to identify trends or areas needing improvement.

o Ensure that the test coverage aligns with the defined acceptance criteria.

o Calculate the defect density, pass/fail ratio, and overall testing effectiveness.

4. Prepare Final Test Report:

o Compile all relevant data, such as test case execution status, defect reports,

and coverage metrics.

o Provide a high-level summary of the testing process, including achievements,

challenges, and lessons learned.

o Offer recommendations for future projects or improvements in the testing

process.

5. Verify Test Environment and Artifacts:

o Ensure that all test environments are cleaned up and that any remaining test

data is properly disposed of.

o Archive test scripts, logs, and results for future reference.

o Ensure all artifacts are stored in a central location, with proper documentation

for easy retrieval.

6. Conduct Test Closure Meeting (Optional):

o Review the testing process with the project team, highlighting successes and

areas of improvement.

o Discuss any unresolved issues and how they should be handled (e.g., deferred

to the next phase or project).

7. Finalize Test Closure Report:

o Summarize the final outcomes, including any residual risks or known issues.

o Include metrics such as test case execution rates, defect counts, and the overall

quality of the product based on testing.

o Confirm the closure of all outstanding defects or tasks.

Test Closure Activities Steps:

1. Test Completion Criteria Review:

• Verify all test cases have been executed, and the test objectives have been met.

• Ensure that critical test cases have been executed, and if any were deferred, document

the reasons.

2. Documentation of Test Results:

• Record the final status of each test case (pass/fail).

• Document the number of defects found, severity, and status of each defect (open,

resolved, closed).

3. Defect and Coverage Analysis:

• Perform defect analysis to identify common issues, trends, or problem areas.

• Calculate metrics such as defect density, pass/fail ratios, and overall effectiveness of

testing.

4. Final Test Report:

• Prepare a final test report summarizing the results of the testing effort.

• Include key sections:

o Executive Summary

o Test Plan Summary

o Test Execution Summary (pass/fail rate)

o Defects Summary

o Lessons Learned and Recommendations

5. Verify Test Environment and Artifacts:

• Archive test scripts, defect logs, and test results.

• Clean up the test environment and ensure all temporary data is removed.

6. Conduct Test Closure Meeting (Optional):

• Hold a meeting with the project team to discuss the testing process and finalize the

closure.

7. Final Test Closure Report:

• Summarize all activities, including metrics, challenges, and any residual risks or open

issues.

• Ensure the report includes all relevant documentation for future reference or audit.

Sample Test Closure Report:

1. Executive Summary:

• Testing Objectives: Verify that the product meets the defined quality standards by

running functional, regression, and performance tests.

• Test Completion: All 200 test cases executed, with 180 passed and 20 failed.

• Defect Status: 15 defects identified, 10 resolved, 5 deferred to future releases.

• Test Coverage: 95% functional coverage achieved.

2. Test Execution Summary:

Test Type Planned Executed Passed Failed

Functional 100 100 90 10

Regression 50 50 45 5

Performance 30 30 30 0

Security 20 20 15 5

3. Defects Summary:

Defect ID Severity Status Description

DEF-101 High Resolved Login screen crashes with invalid input

DEF-102 Medium Resolved Missing translations in UI

DEF-103 Low Deferred UI alignment issues on mobile view

4. Test Coverage Analysis:

• Functional Coverage: 95%

• Regression Coverage: 90%

• Security Testing Coverage: 100%

• Performance Testing Coverage: 100%

5. Lessons Learned:

• Challenges:

o Performance testing took longer than expected due to high server load.

o Some defect fixes were delayed, impacting the final testing phase.

• Recommendations for Future Projects:

o Improve defect triage and fix cycle for critical issues.

o Ensure adequate resources for performance and security testing early in the

development cycle.

6. Outstanding Risks:

• Five defects are deferred to the next release, with no immediate risk to the product's

release.

Sample Test Closure Activity Checklist:

Activity Status Notes

Review Test Completion Criteria Completed All tests executed

Document Test Results Completed Results documented

Perform Defect Analysis Completed Defects analyzed and tracked

Prepare Final Test Report Completed Report submitted to stakeholders

Archive Test Artifacts and Clean Up Completed All test data archived

Conduct Test Closure Meeting Optional Meeting held with team

Finalize Test Closure Report Completed Report finalized and shared

Results:

By performing test closure activities, including detailed documentation, defect analysis, and

preparation of a final test report, the testing team ensured that the project was properly

concluded. The closure report provided valuable insights into the effectiveness of the testing

process, highlighted any risks or unresolved issues, and offered recommendations