**UNIVERSITY SCHOOL OF INFORMATION &**

**COMMUNICATION TECHNOLOGY (USICT)**

# **GURU GOBIND SINGH INDRAPRASTHA**

# **UNIVERSITY**

## **Sector 16 C, Dwarka, Delhi, 110078**



**Software Testing & Quality Assurance**

#### (IT 662)

**Submitted To: Submitted By:**

**Ms Heena Kwatra Nabin Kumar Pal**

**Enroll no.:07016404524**

**MCA(SE) Sem-2**

**Section: 2**

**Index**

| **Sr. No.** | **Title of the Practical** | **Testing Technique** | **Date** | **Signature** |
| --- | --- | --- | --- | --- |
| 1. | Write test cases using **Boundary Value Analysis** for a numeric input field. | Boundary Value Analysis |  |  |
| 2. | Perform **Equivalence Partitioning** on a login form (username and password). | Black Box Testing |  |  |
| 3. | Design test cases using **Decision Table Testing** for online payment processing. | Decision Table Testing |  |  |
| 4. | Conduct **State Transition Testing** for a vending machine system. | Black Box Testing |  |  |
| 5. | Implement **Path Coverage Testing** on a function with multiple loops and decisions. | White Box Testing |  |  |
| 6. | Apply **Statement Coverage Testing** to a basic calculator program. | White Box Testing |  |  |
| 7. | Create test scenarios for a registration form using **Cause-Effect Graphing**. | Black Box Testing |  |  |
| 8. | Develop test cases to check exception handling in a file-handling module. | White Box Testing |  |  |
| 9. | Perform **Error Guessing** to test a form submission system. | Black Box Testing |  |  |
| 10. | Write unit test cases for a sorting algorithm using assertions. | White Box Testing (Unit) |  |  |
| 11. | Design test cases using **Use Case Testing** for a library management system. | Black Box Testing |  |  |
| 12. | Perform integration testing for two interacting modules (e.g., login & dashboard). | White Box / Gray Box Testing |  |  |
| 13. | Create test data and validate input for boundary values of age (e.g., 18-60). | Boundary Value Analysis |  |  |
| 14. | Automate test cases using a simple testing tool (e.g., Selenium, JUnit). | Tool-based Testing |  |  |
| 15. | Prepare a test report including bug list and test summary for a sample project. | Test Documentation |  |  |

**Program No.1**

**Q.1** Write test cases using **Boundary Value Analysis** for a numeric input field.

**Explanation:**

Boundary Value Analysis (BVA) is a software testing technique that focuses on testing the boundaries of input values. For a numeric input field, you would test the minimum and maximum values, as well as values just outside these boundaries.

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter a number between 1 and 100: ";

cin >> num;

if (num >= 1 && num <= 100) {

cout << "Valid input." << endl;

} else {

cout << "Invalid input." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Input** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | 0 | Invalid input | Invalid input |
| TC2 | 1 | Valid input | Valid input |
| TC3 | 50 | Valid input | Valid input |
| TC4 | 100 | Valid input | Valid input |
| TC5 | 101 | Invalid input | Invalid input |

**Program No.2**

**Q.2** Perform **Equivalence Partitioning** on a login form (username and password).

**Explanation:**

Equivalence Partitioning is a software testing technique that divides the input data into partitions (equivalence classes) and tests one value from each partition. For a login form, you would test valid and invalid combinations of username and password.

**C++ Code:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string username, password;

cout << "Enter username: ";

cin >> username;

cout << "Enter password: ";

cin >> password;

if (username == "admin" && password == "password") {

cout << "Login successful." << endl;

} else {

cout << "Login failed." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Username** | **Password** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | admin | password | Login successful | Login successful |
| TC2 | admin | wrongpwd | Login failed | Login failed |
| TC3 | user | password | Login failed | Login failed |
| TC4 | user | wrongpwd | Login failed | Login failed |

**Program No.3**

**Q.3** Design test cases using **Decision Table Testing** for online payment processing.

**Explanation:**

Decision Table Testing is a software testing technique that uses a table to represent combinations of inputs and their corresponding outputs. For online payment processing, you would test different combinations of payment methods and amounts.

**C++ Code:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string paymentMethod;

double amount;

cout << "Enter payment method (credit/debit): ";

cin >> paymentMethod;

cout << "Enter amount: ";

cin >> amount;

if ((paymentMethod == "credit" || paymentMethod == "debit") && amount > 0) {

cout << "Payment successful." << endl;

} else {

cout << "Payment failed." << endl;

}

return 0;}

**Test Cases Table:**

| **Test Case ID** | **Payment Method** | **Amount** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | credit | 100 | Payment successful | Payment successful |
| TC2 | debit | 100 | Payment successful | Payment successful |
| TC3 | cash | 100 | Payment failed | Payment failed |
| TC4 | credit | -50 | Payment failed | Payment failed |

**Program No.4**

**Q.4** Conduct **State Transition Testing** for a vending machine system.

**Explanation:**

State Transition Testing is a software testing technique that tests the transitions between different states of a system. For a vending machine, you would test the transitions between states like "waiting for coin," "coin inserted," and "item selected."

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

int state = 0;

int input;

while (state != 3) {

cout << "Enter input (1: Insert Coin, 2: Select Item, 3: Exit): ";

cin >> input;

switch (input) {

case 1:

state = 1;

cout << "Coin inserted." << endl;

break;

case 2:

if (state == 1) {

state = 2;

cout << "Item selected." << endl;

} else {

cout << "Insert coin first." << endl;

}

break;

case 3:

state = 3;

cout << "Exiting." << endl;

break;

default:

cout << "Invalid input." << endl;

}

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Input Sequence** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | 1, 2, 3 | Coin inserted, Item selected, Exiting | Coin inserted, Item selected, Exiting |
| TC2 | 2, 1, 3 | Insert coin first, Coin inserted, Exiting | Insert coin first, Coin inserted, Exiting |
| TC3 | 1, 3 | Coin inserted, Exiting | Coin inserted, Exiting |

**Program No.5**

**Q.5** Implement **Path Coverage Testing** on a function with multiple loops and decisions.

**Explanation:**

Path Coverage Testing is a software testing technique that ensures all possible paths through a program are tested. For a function with multiple loops and decisions, you would test different paths based on the conditions.

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

int x, y;

cout << "Enter x: ";

cin >> x;

cout << "Enter y: ";

cin >> y;

if (x > 0) {

if (y > 0) {

cout << "Both x and y are positive." << endl;

} else {

cout << "x is positive, y is not." << endl;

}

} else {

if (y > 0) {

cout << "x is not positive, y is." << endl;

} else {

cout << "Both x and y are not positive." << endl;

}

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **x** | **y** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | 1 | 1 | Both x and y are positive | Both x and y are positive |
| TC2 | 1 | -1 | x is positive, y is not | x is positive, y is not |
| TC3 | -1 | 1 | x is not positive, y is | x is not positive, y is |
| TC4 | -1 | -1 | Both x and y are not positive | Both x and y are not positive |

**Program No.6**

**Q.6** Apply **Statement Coverage Testing** to a basic calculator program.

**Explanation:**

Statement Coverage Testing is a software testing technique that ensures each statement in the program is executed at least once. For a basic calculator, you would test different operations and inputs.

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

char op;

double num1, num2;

cout << "Enter operator (+, -, \*, /): ";

cin >> op;

cout << "Enter two operands: ";

cin >> num1 >> num2;

switch (op) {

case '+':

cout << "Result: " << num1 + num2 << endl;

break;

case '-':

cout << "Result: " << num1 - num2 << endl;

break;

case '\*':

cout << "Result: " << num1 \* num2 << endl;

break;

case '/':

if (num2 != 0) {

cout << "Result: " << num1 / num2 << endl;

} else {

cout << "Division by zero error." << endl;

}

break;

default:

cout << "Invalid operator." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Operator** | **Operand 1** | **Operand 2** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- | --- |
| TC1 | + | 5 | 3 | Result: 8 | Result: 8 |
| TC2 | - | 5 | 3 | Result: 2 | Result: 2 |
| TC3 | \* | 5 | 3 | Result: 15 | Result: 15 |
| TC4 | / | 6 | 3 | Result: 2 | Result: 2 |
| TC5 | / | 6 | 0 | Division by zero error | Division by zero error |
| TC6 | % | 5 | 3 | Invalid operator | Invalid operator |

**Program No.7**

**Q.7** Create test scenarios for a registration form using **Cause-Effect Graphing**.

**Explanation:**

Cause-Effect Graphing is a software testing technique that uses a graph to represent the relationships between inputs (causes) and outputs (effects). For a registration form, you would test different combinations of inputs like name, email, and age.

**C++ Code:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name, email;

int age;

cout << "Enter name: ";

cin >> name;

cout << "Enter email: ";

cin >> email;

cout << "Enter age: ";

cin >> age;

if (!name.empty() && email.find('@') != string::npos && age >= 18 && age <= 60) {

cout << "Registration successful." << endl;

} else {

cout << "Registration failed." << endl;}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Name** | **Email** | **Age** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- | --- |
| TC1 | John | [john@gmail.com](mailto:john@example.com" \t "_blank) | 25 | Registration successful | Registration successful |
| TC2 |  | [john@gmail.com](mailto:john@example.com" \t "_blank) | 25 | Registration failed | Registration failed |
| TC3 | John | john.com | 25 | Registration failed | Registration failed |
| TC4 | John | [john@gmail.com](mailto:john@example.com" \t "_blank) | 17 | Registration failed | Registration failed |
| TC5 | John | [john@gmail.com](mailto:john@example.com" \t "_blank) | 61 | Registration failed | Registration failed |

**Program No.8**

**Q.8** Develop test cases to check exception handling in a file-handling module.

**Explanation:**

Exception Handling is a software testing technique that ensures the program handles exceptions gracefully. For a file-handling module, you would test different scenarios like file not found, read/write errors, etc.

**C++ Code:**

#include <iostream>

#include <fstream>

using namespace std;

int main() {

ifstream file("example.txt");

if (file.is\_open()) {

cout << "File opened successfully." << endl;

file.close();

} else {

cout << "Unable to open file." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **File Existence** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | File exists | File opened successfully | File opened successfully |
| TC2 | File does not exist | Unable to open file | Unable to open file |

**Program No.9**

**Q.9** Perform **Error Guessing** to test a form submission system.

**Explanation:**

Error Guessing is a software testing technique that involves guessing possible errors and testing for them. For a form submission system, you would test for common errors like missing fields, invalid input formats, etc.

**C++ Code:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name, email;

cout << "Enter name: ";

cin >> name;

cout << "Enter email: ";

cin >> email;

if (!name.empty() && email.find('@') != string::npos) {

cout << "Form submission successful." << endl;

} else {

cout << "Form submission failed." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Name** | **Email** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | John | [john@example.com](mailto:john@example.com" \t "_blank) | Form submission successful | Form submission successful |
| TC2 |  | [john@example.com](mailto:john@example.com" \t "_blank) | Form submission failed | Form submission failed |
| TC3 | John | john.com | Form submission failed | Form submission failed |

**Program No.10**

**Q.10** Write unit test cases for a sorting algorithm using assertions.

**Explanation:**

Unit Testing is a software testing technique that tests individual components of a program. For a sorting algorithm, you would write test cases to ensure the algorithm sorts correctly.

**C++ Code:**

#include <iostream>

#include <vector>

#include <cassert>

using namespace std;

void bubbleSort(vector<int>& arr) {

int n = arr.size();

for (int i = 0; i < n-1; i++) {

for (int j = 0; j < n-i-1; j++) {

if (arr[j] > arr[j+1]) {

swap(arr[j], arr[j+1]);

}

}

}

}

int main() {

vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

bubbleSort(arr);

vector<int> expected = {11, 12, 22, 25, 34, 64, 90};

assert(arr == expected);

cout << "Test passed." << endl;

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Input Array** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | {64, 34, 25, 12, 22, 11, 90} | {11, 12, 22, 25, 34, 64, 90} | {11, 12, 22, 25, 34, 64, 90} |

**Program No.11**

**Q.11** Design test cases using **Use Case Testing** for a library management system.

**Explanation:**

Use Case Testing is a software testing technique that tests the system based on use cases. For a library management system, you would test different use cases like borrowing a book, returning a book, etc.

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

int choice;

cout << "Enter choice (1: Borrow Book, 2: Return Book): ";

cin >> choice;

switch (choice) {

case 1:

cout << "Book borrowed successfully." << endl;

break;

case 2:

cout << "Book returned successfully." << endl;

break;

default:

cout << "Invalid choice." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Choice** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | 1 | Book borrowed successfully | Book borrowed successfully |
| TC2 | 2 | Book returned successfully | Book returned successfully |
| TC3 | 3 | Invalid choice | Invalid choice |

**Program No.12**

**Q.12** Perform integration testing for two interacting modules (e.g., login & dashboard).

**Explanation:**

Integration Testing is a software testing technique that tests the interaction between two or more modules. For a login and dashboard system, you would test the integration between the login module and the dashboard module.

**C++ Code:**

#include <iostream>

#include <string>

using namespace std;

bool login(string username, string password) {

return username == "admin" && password == "password";

}

void dashboard() {

cout << "Welcome to the dashboard." << endl;

}

int main() {

string username, password;

cout << "Enter username: ";

cin >> username;

cout << "Enter password: ";

cin >> password;

if (login(username, password)) {

dashboard();

} else {

cout << "Login failed." << endl;}

return 0;}

**Test Cases Table:**

| **Test Case ID** | **Username** | **Password** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | admin | password | Welcome to the dashboard | Welcome to the dashboard |
| TC2 | admin | wrongpwd | Login failed | Login failed |
| TC3 | user | password | Login failed | Login failed |

**Program No.13**

**Q.13** Create test data and validate input for boundary values of age (e.g., 18-60).

**Explanation:**

Boundary Value Analysis is used to test the boundary values of input data. For age, you would test values just below, at, and just above the boundary values.

**C++ Code:**

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter age: ";

cin >> age;

if (age >= 18 && age <= 60) {

cout << "Valid age." << endl;

} else {

cout << "Invalid age." << endl;

}

return 0;

}

**Test Cases Table:**

| **Test Case ID** | **Age** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- |
| TC1 | 17 | Invalid age | Invalid age |
| TC2 | 18 | Valid age | Valid age |
| TC3 | 30 | Valid age | Valid age |
| TC4 | 60 | Valid age | Valid age |
| TC5 | 61 | Invalid age | Invalid age |

**Program No.14**

**Q.14** Automate test cases using a simple testing tool (e.g., Selenium, JUnit).

**Explanation:**

Automated Testing is a software testing technique that uses tools to automate the testing process. For a simple testing tool like Selenium or JUnit, you would write test scripts to automate the testing of web applications or Java programs.

**Example (JUnit):**

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

assertEquals(8, calc.add(5, 3));

}

@Test

public void testSubtract() {

Calculator calc = new Calculator();

assertEquals(2, calc.subtract(5, 3));

}

}

**Test Cases Table:**

| **Test Case ID** | **Method** | **Input** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| TC1 | add | 5, 3 | 8 | 8 |
| TC2 | subtract | 5, 3 | 2 | 2 |

**Program No.15**

**Q.15** Prepare a test report including bug list and test summary for a sample project.

**Test Report for E-Library Management System**

**1. Project Overview**

**Project Name:** E-Library Management System

**Project Description:**

The E-Library Management System is a web-based application designed to manage library operations efficiently. It includes features such as user registration, login, book search, book borrowing, and book returning. The system aims to streamline library management and enhance user experience.

**Test Environment:**

* **Operating System:** Windows 10
* **Browser:** Google Chrome, Firefox
* **Database:** MySQL
* **Server:** Apache
* **Programming Languages:** HTML, CSS, JavaScript, PHP

1. **Test Summary**

**Test Objectives:**

* Ensure the application functions as specified.
* Identify and document any defects or issues.
* Validate user interface and user experience.

**Test Scope:**

* User registration and login.
* Book search functionality.
* Book borrowing and returning.
* Dashboard functionality.
* Error handling and validation.

**Test Cases Executed:**

* Total Test Cases: 25
* Test Cases Passed: 22
* Test Cases Failed: 3

**Testing Tools:**

* **Manual Testing:** Selenium WebDriver
* **Automated Testing:** JUnit

**Testing Period:**

* Start Date: April 1, 2025
* End Date: April 15, 2025

**3. Bug List**

**Critical Bugs:**

| **Bug ID** | **Description** | **Severity** | **Status** | **Assigned To** |
| --- | --- | --- | --- | --- |
| BUG001 | User unable to log in with correct credentials | High | Open | John Doe |
| BUG002 | Division by zero error in book rating system | High | Open | Jane Smith |

**Major Bugs:**

| **Bug ID** | **Description** | **Severity** | **Status** | **Assigned To** |
| --- | --- | --- | --- | --- |
| BUG003 | Incorrect book return date calculation | Medium | Open | Alice Johnson |
| BUG004 | Dashboard not displaying correct user information | Medium | Open | Bob Lee |

**Minor Bugs:**

| **Bug ID** | **Description** | **Severity** | **Status** | **Assigned To** |
| --- | --- | --- | --- | --- |
| BUG005 | Typos in user registration form | Low | Open | Chris Brown |
| BUG006 | Inconsistent font sizes on different pages | Low | Open | Emily Davis |

**4. Test Summary**

**Test Cases Executed:**

* **User Registration and Login:**
  + TC001: Successful user registration.
  + TC002: Failed user registration with invalid email.
  + TC003: Successful user login.
  + TC004: Failed user login with incorrect password.
* **Book Search Functionality:**
  + TC005: Successful book search by title.
  + TC006: Successful book search by author.
  + TC007: No results found for invalid search query.
* **Book Borrowing and Returning:**
  + TC008: Successful book borrowing.
  + TC009: Failed book borrowing due to insufficient stock.
  + TC010: Successful book returning.
  + TC011: Failed book returning due to incorrect book ID.
* **Dashboard Functionality:**
  + TC012: Dashboard displays correct user information.
  + TC013: Dashboard displays correct book borrowing history.
* **Error Handling and Validation:**
  + TC014: Error message displayed for invalid input.
  + TC015: Error message displayed for division by zero in book rating system.

**Test Cases Passed:**

* TC001, TC003, TC005, TC007, TC008, TC010, TC012, TC014

**Test Cases Failed:**

* TC002, TC004, TC006, TC009, TC011, TC013, TC015

**Issues Identified:**

* User unable to log in with correct credentials (BUG001).
* Division by zero error in book rating system (BUG002).
* Incorrect book return date calculation (BUG003).
* Dashboard not displaying correct user information (BUG004).
* Typos in user registration form (BUG005).
* Inconsistent font sizes on different pages (BUG006).

**Test Case Execution Table for E-Library Management System**

| **Test Case ID** | **Test Case Description** | **Expected Result** | **Actual Result** | **Status (Pass/Fail)** |
| --- | --- | --- | --- | --- |
| TC001 | Successful user registration with valid details | User registration successful | User registration successful | Pass |
| TC002 | Failed user registration with invalid email | User registration fails with error message | User registration fails with error message | Pass |
| TC003 | Successful user login with correct credentials | User login successful | User login successful | Pass |
| TC004 | Failed user login with incorrect password | User login fails with error message | User login fails with error message | Pass |
| TC005 | Successful book search by title | Correct book details displayed | Correct book details displayed | Pass |
| TC006 | Successful book search by author | Correct book details displayed | Correct book details displayed | Pass |
| TC007 | No results found for invalid search query | No results found message displayed | No results found message displayed | Pass |
| TC008 | Successful book borrowing | Book borrowed successfully | Book borrowed successfully | Pass |
| TC009 | Failed book borrowing due to insufficient stock | Error message displayed for insufficient stock | Error message displayed for insufficient stock | Pass |
| TC010 | Successful book returning | Book returned successfully | Book returned successfully | Pass |
| TC011 | Failed book returning due to incorrect book ID | Error message displayed for incorrect book ID | Error message displayed for incorrect book ID | Pass |
| TC012 | Dashboard displays correct user information | Correct user information displayed | Correct user information displayed | Pass |
| TC013 | Dashboard displays correct book borrowing history | Correct book borrowing history displayed | Correct book borrowing history displayed | Pass |
| TC014 | Error message displayed for invalid input | Error message displayed | Error message displayed | Pass |
| TC015 | Error message displayed for division by zero in book rating system | Error message displayed | Error message displayed | Pass |