

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY

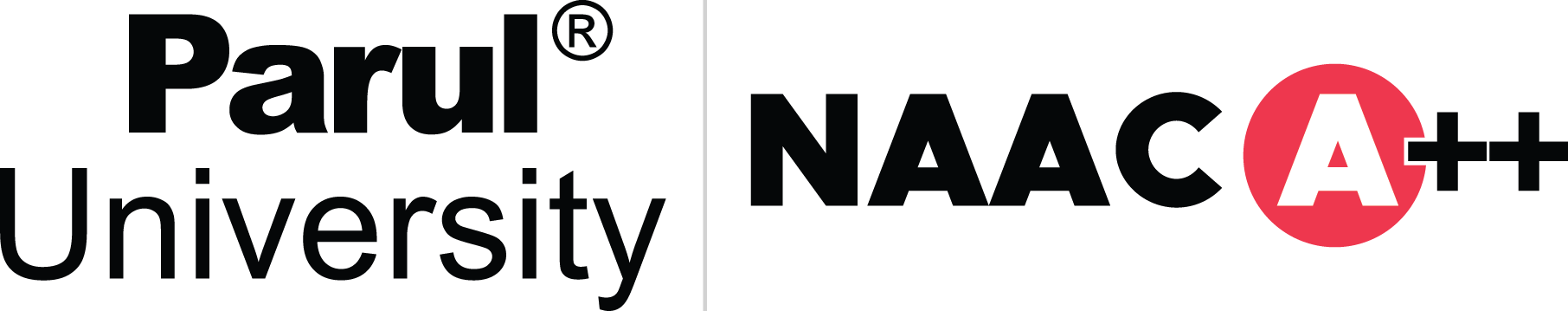
BACHELOR OF TECHNOLOGY

SOFTWARE TESTING AND QUALITY ASSURANCE LABORATORY

(303105378)

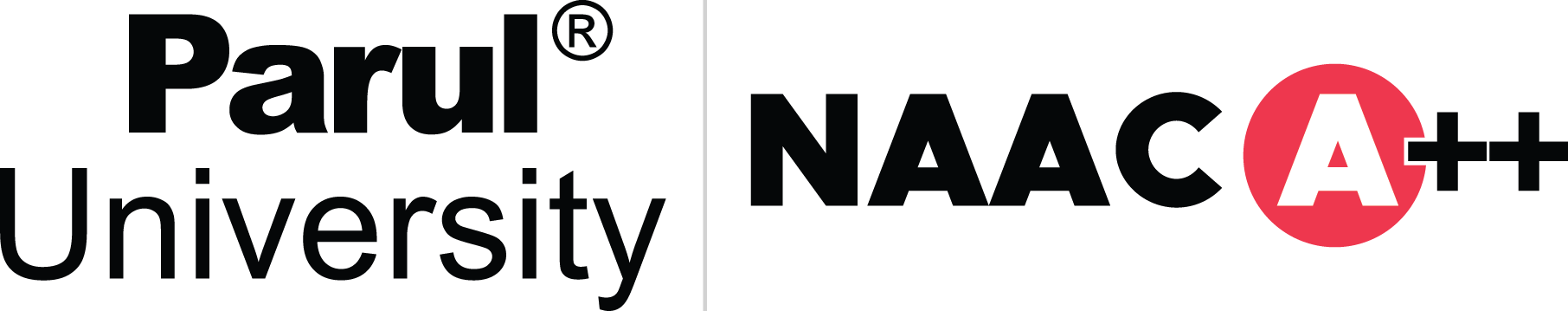
IV SEMESTER

Computer Science & Engineering Department



Laboratory Manual

Session 2025-26



Faculty of Engineering & Technology

Subject Name : STQA Laboratory

Subject Code : 303105378

B.Tech CSE Year 4th Semester 7th

CERTIFICATE

This is to Certify that

Mr./Ms. …………………………………………………………………………………………………...

With enrolment no. ………………………………….has successfully completed his/her

Laboratory experiments in **Software Testing and Quality Assurance Laboratory (303105378)** From the department of …………………………………………………………..… during the academic year ………………………



Head of department: ……………………….

Date of Submission : ………………………. Staff In Charge: ……………………….

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr. No** | **Experimental Title** | **Page No** | | **Date of Completion** | **Sign** | **Marks (out of 10)** |
| **From** | **To** |
| 1. | Design test cases using Boundary value analysis |  |  |  |  |  |
| 2. | Design test cases using Equivalence class partitioning |  |  |  |  |  |
| 3. | Design independent paths by calculating cyclometic complexity using date problem |  |  |  |  |  |
| 4. | Design test cases using Decision table. |  |  |  |  |  |
| 5. | Design independent paths by taking DD path using date problem. |  |  |  |  |  |
| 6. | Understand The Automation Testing Approach (Theory Concept). |  |  |  |  |  |
| 7. | Using Selenium IDE, Write a test suite containing minimum 4 test cases |  |  |  |  |  |
| 8. | Install Selenium server and demonstrate it using a script in Java/PHP. |  |  |  |  |  |
| 9. | Write and test a program to login a specific web page. |  |  |  |  |  |
| 10. | Write and test a program to provide total number of objects present / available on the page.. |  |  |  |  |  |
| 11. | Write and test a program to update 10 student records into table into Excel file |  |  |  |  |  |

**TABLE OF CONTENT**

PRACTICAL - 1

**AIM** - Design test cases using Boundary value analysis.

**Theory: -**

1. **Boundary Value** –

The term Boundary Value refers to the limits or edges of an input domain. In software testing, it focuses on checking the behavior of the program at the extremes of input ranges.

Common words and concepts related to Boundary Value include:

* Minimum value (Lower limit)
* Maximum value (Upper limit)
* Just below minimum (Minimum - 1)
* Just above minimum (Minimum + 1)
* Just below maximum (Maximum - 1)
* Just above maximum (Maximum + 1)
* Valid range (Between min and max)
* Invalid inputs (Outside the defined range)

**2. Boundary Value Analysis –**

**Boundary Value Analysis (BVA)** is a black-box test design technique that involves testing at the boundaries between partitions. The idea is that if a system works well at boundary values, it will likely handle other values within the range correctly.

**Definition:**  
Boundary Value Analysis tests the values at the **edge of input ranges**, as these are the points where defects are most likely to occur.

**Formula for Boundary Value Test Cases:**  
For a given input range [min, max], we test:

* min - 1 → Just below minimum
* min → Minimum
* min + 1 → Just above minimum
* max - 1 → Just below maximum
* max → Maximum
* max + 1 → Just above maximum

So, the **Test Inputs = {min - 1, min, min + 1, max - 1, max, max + 1}**

**3. Boundary Value Testing – Practical Implementation**

**Problem Statement:-** Validate if a given age is within the range **18 to 60 (inclusive)**.

**Function to Test:-**

def is\_valid\_age(age):

if 18 <= age <= 60:

return "Valid"

else:

return "Invalid"

**Test Cases Based on Boundary Value Analysis:**

| **Test Case No.** | **Input Age** | **Description** | **Expected Output** |
| --- | --- | --- | --- |
| TC1 | 17 | Just below minimum | Invalid |
| TC2 | 18 | Minimum boundary | Valid |
| TC3 | 19 | Just above minimum | Valid |
| TC4 | 59 | Just below maximum | Valid |
| TC5 | 60 | Maximum boundary | Valid |
| TC6 | 61 | Just above maximum | Invalid |

**PROGRAM:-**

***def*** *is\_valid\_age(****age****):*

***if*** *18* ***<=*** *age* ***<=*** *60:*

***return*** *"Valid"*

***else****:*

***return*** *"Invalid"*

*#Define test cases*

*test\_cases* ***=*** *[*

*(18, "Valid"), # Minium boundary*

*(17, "Invalid"), # Just below minimum boundary*

*(19, "Valid"), # Just above minimum boundary*

*(59, "Valid"), # Just below maximum boundary*

*(60, "Valid"), # Maximum boundary*

*(61, "Invalid") # Just above maximum boundary*

*]*

*# Run test cases*

***for*** *i, (age, expected)* ***in*** *enumerate(test\_cases):*

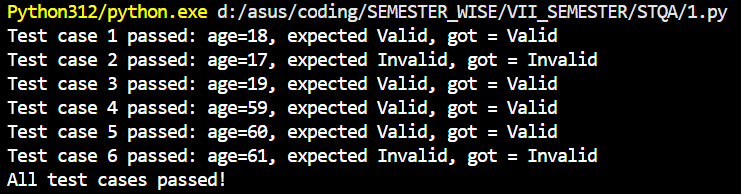
*result* ***=*** *is\_valid\_age(age)*

***assert*** *result* ***==*** *expected, f"Test case {i****+****1} failed: age={age}, expected {expected}"*

*print(f"Test case {i****+****1} passed: age={age}, expected {expected}, got = {result}")*

*print("All test cases passed!")*

**OUTPUT :**

****

**Conclusion:**

Using Boundary Value Analysis, we identified key edge cases for the age validation function. Testing inputs at and around the minimum and maximum boundaries helped ensure the function behaves as expected in both valid and invalid cases.

PRACTICAL - 2

**AIM** - Design test cases using Equivalence class partitioning.

**Theory: -**

**What is Equivalence Class Partitioning:**

Equivalence Class Partitioning is a software testing technique where input data is divided into groups or "equivalence classes" based on the assumption that each value within a class will be processed similarly by the software.

By testing a single representative value from each class, testers can achieve maximum test coverage with minimal test cases.

Equivalence classes are derived based on the input conditions, and each input condition may have:

One valid class (acceptable input),

One or more invalid classes (unacceptable input).

This technique is applicable for both input domain testing and output domain testing.

What is Equivalence Class Partitioning Testing:

Equivalence Class Partitioning Testing is the application of the equivalence partitioning technique to design test cases. It focuses on selecting representative inputs from each class to effectively uncover defects.

**Key Characteristics:**

Reduces the total number of test cases.

Avoids redundancy in testing.

Ensures proper coverage of valid and invalid input ranges.

Applicable to all types of inputs: numbers, characters, strings, dates, etc.

**Steps to Perform ECP Testing**:

1.Identify the input domain or conditions.

2.Determine valid and invalid equivalence classes.

3.Select representative values from each class.

4.Design test cases to cover each class.

3. Equivalence Class Testing

Problem Statement:

We need to validate the age of a person, and we will consider the valid age range to be 18 to 60 (inclusive). Any age outside this range is considered invalid.

**Function to Test:**

def is\_valid\_age(age):

if 18 <= age <= 60:

return "Valid"

else:

return "Invalid"

**Equivalence Classes:**

* Valid Equivalence Class (18 ≤ age ≤ 60):
  + Example: Any age between 18 and 60, like 30.
* Invalid Equivalence Class (age < 18):
  + Example: Any age below 18, like 10 or 17.
* Invalid Equivalence Class (age > 60):
  + Example: Any age above 60, like 61 or 70.

**Test Cases Based on Equivalence Class Partitioning:**

| **Test Case No.** | **Input Age** | **Equivalence Class** | **Expected Output** |
| --- | --- | --- | --- |
| TC1 | 30 | Valid (Within valid range) | Valid |
| TC2 | 17 | Invalid (Below valid range) | Invalid |
| TC3 | 10 | Invalid (Well below range) | Invalid |
| TC4 | 61 | Invalid (Above valid range) | Invalid |
| TC5 | 70 | Invalid (Well above range) | Invalid |

**CODE-**

*# Define the function*

**def** is\_valid\_age(**age**):

**if** 18 **<=** age **<=** 60:

**return** "Valid"

**else**:

**return** "Invalid"

*# Define test cases using Equivalence Class Partitioning*

test\_cases **=** [

(30, "Valid"), *# Valid range middle value*

(17, "Invalid"), *# Just below the valid range*

(10, "Invalid"), *# Well below the valid range*

(61, "Invalid"), *# Just above the valid range*

(70, "Invalid"), *# Well above the valid range*

]

*# Run test cases*

**for** i, (age, expected) **in** enumerate(test\_cases):

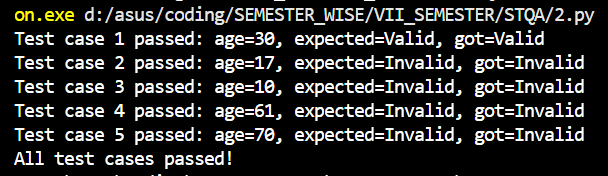
    result **=** is\_valid\_age(age)

**assert** result **==** expected, f"Test case {i**+**1} failed: age={age}, expected={expected}, got={result}"

    print(f"Test case {i**+**1} passed: age={age}, expected={expected}, got={result}")

print("All test cases passed!")

**OUTPUT –**



**Conclusion:**

Using **Equivalence Class Partitioning (ECP)**, we divided the input domain into **valid and invalid classes**. By testing values from each equivalence class, we verified that the function handles both valid and invalid input correctly.