Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

SOFTWARE TESTING AND QUALITY ASSURANCE
(3033105377)

NAME: - PANDEY SANJIT VINOD
ENROLLMENT NO: - 2203051050314
DIVISION: - 7A13
ROLL NO : 25

SEMESTER VII
Computer Science and Engineering Department

Laboratory Manual

Session:2025-26

CERTIFICATE

Ms. ***KRITIKA JAIN*** *with Enrollment No.* ***2203051050314*** *has successfully completed his/her laboratory experiments* ***Software Testing and Quality Assurance (303105377)*** *from the department of* ***Computer Science and Engineering*** *during the academic year* ***2025-2026.***

Date of Submission …….. **Staff In charge ……..**

Head of Department……

INDEX

Sr. No	Experiment Title	Page No		Start Date	Completion Date	Sign	Marks/ 10
		From	To				
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

Experiment No: 1

Aim: - Design test cases using Boundary Value Analysis (BVA).

Solution:-

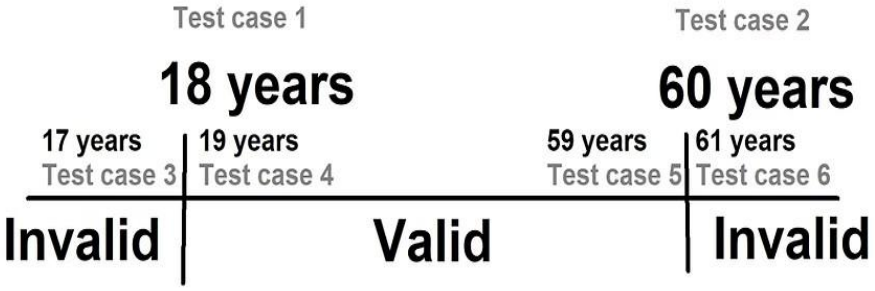
Theory:

Boundary Value Analysis (BVA) is a black-box test design technique that focuses on the **boundaries of input values**. The core idea is simple: errors often occur at the **edges** of input domains rather than in the middle. So instead of testing all possible inputs, BVA tests the values at the edge—where things tend to go wrong.

Boundary Value Analysis focuses on values at the edge of valid ranges where bugs frequently occur.
The technique assumes that if a system works correctly at its boundaries, it likely works throughout the input range.

For a valid age range of 18–60, BVA would include:

- Just below the lower boundary: 17
- At the lower boundary: 18
- Just above the lower boundary: 19
- Just below the upper boundary: 59
- At the upper boundary: 60
- Just above the upper boundary: 61



Problem Statement

Create a program to check if a given integer input x is within the **valid range of 18 to 60** (inclusive). Use **BVA** to design your test cases.

Formula :

- **min - 1, min, min + 1, max - 1, max, max + 1**

Test Case Design (Using BVA)

TestCase ID	Input	Expected Output	Type
TC01	9	Invalid Input	Just below LB
TC02	10	Valid Input	At LB
TC03	11	Valid Input	Just above LB
TC04	99	Valid Input	Just below UB
TC05	100	Valid Input	At UB
TC06	101	Invalid Input	Just above UB

Code:

```python
def is_valid_age(age):
    if 18 <= age <= 60:
        return "Valid"
    else:
        return "Invalid"


test_cases = [
    (30, "Valid"),
    (17, "Invalid"),
    (10, "Invalid"),
    (61, "Invalid"),
    (70, "Invalid"),
]

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:

```
Test case 1 passed: age=30, expected=Valid, got=Valid
Test case 2 passed: age=17, expected=Invalid, got=Invalid
Test case 3 passed: age=10, expected=Invalid, got=Invalid
Test case 4 passed: age=61, expected=Invalid, got=Invalid
Test case 5 passed: age=70, expected=Invalid, got=Invalid
All test cases passed!
```

Analysis

- All test cases passed successfully.
- The function correctly handles:
- A **mid-range valid input** (30)
- Values **below** the valid lower bound (10, 17)
- Values **above** the valid upper bound (61, 70)
- However, **no test cases** exist for **boundary edges**: 18 (lower bound) and 60 (upper bound).

- These are **critical for Boundary Value Analysis (BVA)**.
- Without testing 18 and 60, it's unclear if the function handles **inclusive edges** properly (even though logically it should).

Conclusion:

The given code defines a function `is_valid_age(age)` that checks if an age is between 18 and 60 (inclusive). If so, it returns "Valid"; otherwise, it returns "Invalid". The script then tests this function with several cases, confirming correct behavior for ages both inside and outside the valid range. If all test cases pass, it prints "All test cases passed!" This confirms that the function correctly validates ages as intended..

Experiment No: 2

Aim: - Design test cases using Equivalence class partitioning.

Solution:-

Equivalence Class Equivalence Class Partitioning is a **black-box testing technique** that organizes test inputs into groups where the system is expected to behave the same. It assumes that testing **one representative** value from each group is enough to verify correct system behavior for that entire group—making your testing **efficient and focused**.

Partitioning Input Domain: Divide all possible inputs into **equivalence classes**—sets of values that are logically treated the same by the software.

Valid Classes: Contain values the system should accept.

Invalid Classes: Contain values the system should reject or handle as errors.

Test Reduction: Instead of testing every single input, you choose **one representative value** from each class, drastically reducing the number of test cases.

Equivalence Class Partitioning, we group input values into **classes** based on expected behavior. Each class will have one **representative value**, unlike BVA which tests multiple points at each boundary.

Class Type	Range of Inputs	Representative Value	Expected Result
Valid Class	18 to 60	30 (or any value in range)	Accepted
Invalid Class (low)	Less than 18	17	Rejected
Invalid Class (high)	Greater than 60	61	Rejected

Equivalence Class Partitioning doesn't require testing *just below* and *just above* boundaries unless you combine it with BVA. You simply check:

- One value **within** the range (like 30)
- One value **below** the range (like 17)
- One value **above** the range (like 61)

Formula :

Total Test Cases = (Number of valid equivalence classes) + (Number of invalid equivalence classes)

This means:

- For each input condition, identify all valid ranges or types.
- Then identify invalid ranges or types that fall outside those bounds.

TestCase table :

Test Case ID	Input	Expected Output	Category
TC01	18	Valid	Lower Boundary
TC02	17	Not Valid	Just Below LB
TC03	19	Valid	Just Above LB
TC04	59	Valid	Just Below UB
TC05	60	Valid	Upper Boundary
TC06	61	Not Valid	Just Above UB
TC07	200	Not Valid	Far Above UB
TC08	87	Not Valid	Above UB
TC09	1	Not Valid	Far Below LB
TC10	9	Not Valid	Below LB
TC11	0	Not Valid	Zero Boundary
TC12	-5	Not Valid	Negative Input

Code:

```python
def is_valid_age(age):
    if 18<=age<=60:
        return "Valid"
    else:
        return "Not Valid"

test_cases=[
    (18,"Valid"),
    (17,"Not Valid"),
    (19,"Valid"),
    (59,"Valid"),
    (60,"Valid"),
    (61,"Not Valid"),
    (200,"Not Valid"),
    (87,"Not Valid"),
    (1,"Not Valid"),
    (9,"Not Valid"),
    (0,"Not Valid"),
    (-5,"Not Valid")
]

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:-

```
Test case 1 passed: age=18, expected=Valid, got=Valid
Test case 2 passed: age=17, expected=Not Valid, got=Not Valid
Test case 3 passed: age=19, expected=Valid, got=Valid
Test case 4 passed: age=59, expected=Valid, got=Valid
Test case 5 passed: age=60, expected=Valid, got=Valid
Test case 6 passed: age=61, expected=Not Valid, got=Not Valid
Test case 7 passed: age=200, expected=Not Valid, got=Not Valid
Test case 8 passed: age=87, expected=Not Valid, got=Not Valid
Test case 9 passed: age=1, expected=Not Valid, got=Not Valid
Test case 10 passed: age=9, expected=Not Valid, got=Not Valid
Test case 11 passed: age=0, expected=Not Valid, got=Not Valid
Test case 12 passed: age=-5, expected=Not Valid, got=Not Valid
All test cases passed!
```

Analysis:

Positives:

- Boundary Value Analysis (BVA) is fully covered.
- Robustness Testing is also well addressed:
  - Includes large upper values (200)
  - Negative values (-5)
  - Zero (0)

Areas for Minor Improvement:

- Could include a midpoint value, like (39, "Valid") or (30, "Valid"), just for completeness.
- Could add a non-integer input test (e.g., "25", 25.5) — if this is a real-world validation scenario.

Conclusion:

The code defines a function `is_valid_age(age)` that checks if an age falls within the inclusive range of 18 to 60. If the age is within this range, the function returns "Valid"; otherwise, it returns "Not Valid". The program thoroughly tests the function with various edge and typical cases, confirming that the function correctly distinguishes valid and invalid ages as required. All test cases pass, demonstrating that the function is robust and behaves as expected for both boundary and out-of-range values.

Experiment No: 3

Aim: - Design independent paths by calculating cyclometric complexity using date problem in python.

Solution:-

Theory

Cyclomatic Complexity is a software metric introduced by **Thomas McCabe** to measure the complexity of a program's control flow.

Purpose:

- Counts **linearly independent paths** in a program.
- Helps determine the **minimum number of test cases** for **100% branch coverage**.
- Assesses **maintainability** and **testability**.

Formulas:

1. **Graph-based:**

$M=E-N+2P$ $M = E - N + 2P$

- E = Edges, N = Nodes, P = Connected components

2. **Decision-based (simpler):**

$M=D+1$ $M = D + 1$

- D = Number of decision points (e.g., if, while, for)

Independent Path:

An **independent path** includes at least one **new edge** not covered by other paths.
✓ Ensures **complete branch coverage** during testing.

Example Calculation:

If a function has 12 decision points:

$M=D+1=12+1=13$ $M = D + 1 = 12 + 1 = \boxed{13}$

→ Minimum **13 test cases** needed for full path coverage.

Here are example independent test paths:

Path	Description	Example Input
1	Valid normal date	10, 3, 2024
2	Year < 1	10, 3, 0
3	Month out of range	10, 13, 2024
4	Day < 1	0, 3, 2024
5	Day > 31 in 31-day month	32, 1, 2024
6	Day > 30 in 30-day month	31, 4, 2024
7	Day > 29 in Feb (leap year)	30, 2, 2024
8	Day = 29 in Feb (leap year)	29, 2, 2024
9	Day > 28 in Feb (non-leap year)	29, 2, 2023
10	Day = 28 in Feb (non-leap year)	28, 2, 2023
11	Month = 2 but not leap year and day = 28	28, 2, 2021
12	Valid edge case: 31 in Dec	31, 12, 2022
13	Leap year test with year = 2000 (div by 400)	29, 2, 2000

Code:-

```
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
def is_valid_date(day, month, year):
```

```
    month_days = [28, 29 if is_leap_year(year) else 31, 31, 30, 31, 30,
                  31, 31, 30, 31, 30, 31]
    if 1 <= month <= 12:
        if 1 <= day <= month_days[month - 1]:
            return "Valid Date"
        else:
            return "Invalid Day for the given month"
    else:
        return "Invalid Month"
day = int(input("Enter day (1-31): "))
month = int(input("Enter month (1-12): "))
year = int(input("Enter year: "))
leap = "Leap Year" if is_leap_year(year) else "Not a Leap Year"
print(f"\nYear Status: {leap}")
```

Output:-

```
Enter day (1-31): 2
Enter month (1-12): 12
Enter year: 2016

Year Status: Leap Year
```

Matrix Table (Condition/Decision Table):-

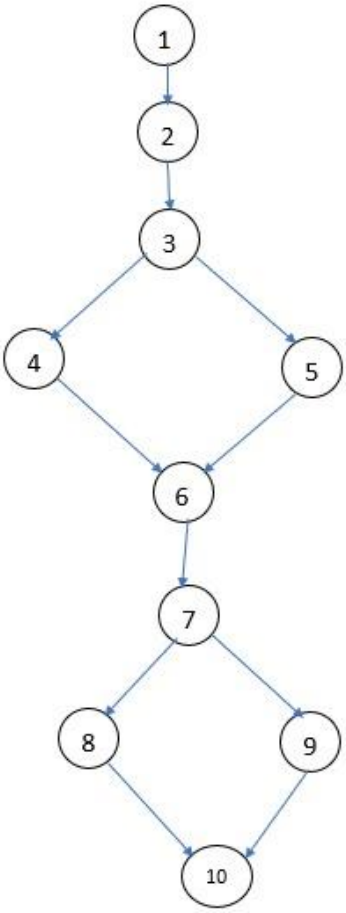
Test Case	Year	is_leap_year	Month	Valid Month?	Day	Max Days in Month	Day Valid?	Output
TC1	2024	Yes	2	Yes	29	29	Yes	Valid Date
TC2	2023	No	2	Yes	29	28	No	Invalid Day for the given month
TC3	2024	Yes	4	Yes	30	30	Yes	Valid Date
TC4	2024	Yes	4	Yes	31	30	No	Invalid Day for

								the given month
TC5	2024	Yes	13	No	10	-	-	Invalid Month
TC6	2024	Yes	0	No	10	-	-	Invalid Month
TC7	2024	Yes	1	Yes	31	31	Yes	Valid Date
TC8	2024	Yes	1	Yes	32	31	No	Invalid Day for the given month
TC9	2000	Yes	2	Yes	29	29	Yes	Valid Date
TC10	1900	No	2	Yes	29	28	No	Invalid Day for the given month

Explanation of Columns:-

Column	**Meaning**
Year	Input year
is_leap_year	Whether the year is leap (based on logic)
Month	Input month
Valid Month?	Whether the month is in range 1-12
Day	Input day
Max Days in Month	Based on month and leap year (from month_days)
Day Valid?	Whether day ≤ max days for that month
Output	Result returned by is_valid_date()

Control Flow Graph:-



Conclusion:-

- Cyclomatic complexity is a useful metric to understand the testing needs of a program.
- For the date validation problem, the complexity is 13, meaning at least 13 test cases are needed for complete path coverage.
- Independent path testing ensures every possible logic condition is tested, increasing code reliability.

Experiment No: 4

Aim: - Design test cases using decision table design independent paths by taking DD path using date problem.

Solution:-

Theory

Software testing ensures that the developed application behaves as expected under different conditions. One of the effective techniques for designing test cases is Decision Table Testing, especially when multiple input conditions affect the output.
In addition, to ensure path coverage, we analyze the program's Decision-to-Decision (DD) paths to generate independent paths and achieve maximum test coverage.

In this problem, we consider a program that validates a given date based on day, month, and year inputs.

Step 1 — Understand the Problem

A date is **valid** if:

1. **Month** ∈ {1..12}
2. **Day** depends on the month:
    - 31 days → Jan, Mar, May, Jul, Aug, Oct, Dec
    - 30 days → Apr, Jun, Sep, Nov
    - Feb → 28 days normally, **29 days in leap years**.
3. **Leap year rule**:
4. If (year % 400 == 0) OR (year % 4 == 0 AND year % 100 != 0) → Leap Year

Step 2 — Decision Table Design

Condition	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**
Month between 1–12?	N	Y	Y	Y	Y	Y
Month has 31 days?	—	Y	N	N	N	N
Month has 30 days?	—	N	Y	N	N	N
February?	—	N	N	Y	Y	Y
Leap year?	—	—	—	Y	N	N
Day ≤ max days?	—	Y	Y	Y	Y	N

Action	Invalid	Valid	Valid	Valid	Valid	Invalid

Explanation of Rules

- **Rule 1** → Invalid month → **Invalid Date**
- **Rule 2** → Month with 31 days and valid day → **Valid**
- **Rule 3** → Month with 30 days and valid day → **Valid**
- **Rule 4** → February in leap year → **Valid if day ≤ 29**
- **Rule 5** → February in non-leap year → **Valid if day ≤ 28**
- **Rule 6** → February day > 29 or any invalid day → **Invalid**

Step 3 — Test Cases from Decision Table

TC No.	Day	Month	Year	Expected Result
TC1	15	13	2023	Invalid (Month > 12)
TC2	31	1	2023	Valid (Jan, 31 days)
TC3	31	4	2023	Invalid (Apr has 30)
TC4	29	2	2024	Valid (Leap year)
TC5	29	2	2023	Invalid (Not leap)
TC6	30	2	2024	Invalid (>29 Feb)
TC7	28	2	2023	Valid
TC8	0	5	2023	Invalid (Day=0)
TC9	31	12	2020	Valid (Dec, 31 days)

Step 4 — Design DD Path (Decision-to-Decision Path)

Decision Points (Nodes)

- **D1:** Check valid month
- **D2:** Check 31-day months
- **D3:** Check 30-day months
- **D4:** Check February
- **D5:** Check Leap Year
- **D6:** Check valid day ranges

Independent Paths

1. **P1:** Invalid month → D1 → Invalid
2. **P2:** Valid 31-day → D1 → D2 → Valid
3. **P3:** Invalid 31-day → D1 → D2 → Invalid
4. **P4:** Valid 30-day → D1 → D3 → Valid
5. **P5:** Invalid 30-day → D1 → D3 → Invalid
6. **P6:** Leap year Feb valid → D1 → D4 → D5 → Valid
7. **P7:** Leap year Feb invalid → D1 → D4 → D5 → Invalid
8. **P8:** Non-leap Feb valid → D1 → D4 → D5 → Valid
9. **P9:** Non-leap Feb invalid → D1 → D4 → D5 → Invalid

Thus, we have **9 independent DD paths**.

Step 5 — Final Output

- Decision Table
- Test Cases
- DD Paths

Experiment No: 5

Aim: - Understand the Automation Testing approach(theory concept).

Solution:-

Theory

Automation Testing is a software testing approach where test cases are executed automatically using specialized testing tools instead of performing them manually.
It is used to improve the efficiency, speed, reliability, and accuracy of the testing process.

In Software Testing & Quality Assurance (QA), automation testing ensures that:

- Repetitive tasks are tested quickly.
- Human errors are minimized.
- Software quality improves.

Why Automation Testing?

Manual testing is time-consuming and error-prone. Automation testing solves these issues by:

- Reducing manual effort.
- Increasing test coverage.
- Executing regression tests quickly.
- Ensuring faster delivery of quality software.

Automation Testing Approach:-

The automation testing approach defines how, when, and where automation should be used in the software testing lifecycle.
It includes the following steps:

Step 1 — Identify Areas for Automation

- Not all tests should be automated.
- Best suited for:
    - Repetitive test cases
    - Regression testing
    - Performance testing
    - Large data-driven tests

Step 2 — Select the Right Automation Tool

- Choose a tool based on:
  - Technology stack of the application.
  - Project requirements.
  - Budget constraints.
- Examples:
  - Selenium, Appium, Cypress → Web & mobile testing
  - JUnit, TestNG → Unit testing
  - JMeter → Performance testing

Step 3 — Define the Automation Scope

- Decide what to automate and what to test manually.
- Focus on high-priority, stable, and repeatable scenarios.

Step 4 — Create the Test Automation Framework

A framework provides guidelines and best practices for automation:

- Types of frameworks:
  - Data-Driven Framework → Uses external test data.
  - Keyword-Driven Framework → Uses keywords for test execution.
  - Hybrid Framework → Combines data-driven + keyword-driven.
  - Behavior-Driven (BDD) → Uses natural language (e.g., Cucumber).

Step 5 — Develop and Execute Test Scripts

- Write automation test scripts using tools like Selenium, JUnit, or TestNG.
- Integrate scripts with the framework.
- Execute tests automatically.

Step 6 — Analyze Results and Generate Reports

- Compare actual results vs expected results.
- Generate reports showing pass/fail status.
- Log defects for failed cases.

Step 7 — Maintain Automation Scripts

- Update scripts when:
  - Application features change.
  - UI layouts are modified.

- Ensures test scripts remain reusable and reliable.

Advantages of Automation Testing:-

Advantage	Description
Faster Execution	Automated scripts run faster than manual testing.
Reusability	Same scripts can be used across multiple test cycles.
Better Accuracy	Eliminates human errors during testing.
Improved Test Coverage	Can run thousands of test cases quickly.
Continuous Testing	Supports CI/CD pipelines for DevOps.

Disadvantages / Limitations:-

Limitation	Description
High Initial Cost	Tools and framework setup require investment.
Not Suitable for All Tests	Exploratory, usability, and ad-hoc testing still need humans.
Maintenance Overhead	Scripts must be updated when the application changes.
Learning Curve	Requires technical skills for scripting and tool usage.

When to Use Automation Testing ?

Automation is most effective when:

- Tests are repetitive.
- Application is stable.
- Regression testing is needed.
- Performance or load testing is required.
- CI/CD pipelines are implemented.

Example:-

Scenario: Credit Card Payment System

- Manual Testing: Enter card details → click Pay → verify success message.
- Automation Testing:
    - Use Selenium to simulate entering details automatically.
    - Check if success or error messages match expected results.

- Run 100+ test cases in seconds instead of hours.

Conclusion:-

The automation testing approach is essential in modern software testing and QA because:

- It reduces manual effort.
- Increases speed and efficiency.
- Improves software reliability.
- Supports continuous integration and delivery.

By using the right tools, frameworks, and strategies, teams can ensure high-quality software in less time and with fewer resources.