

# PRACTICAL: 01

## **Aim:** Create test cases using boundary value analysis.

**Boundary value analysis:**

Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behaviour at the edge of the equivalence partition is more likely to be incorrect than the behaviour within the partition, so boundaries are an area where testing is likely to yield defects.

It checks for the input values near the boundary that have a higher chance of error. Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition.

1. **Date format check**

If Date should be in duration of 10 days i.e 20-11-22 to 30-11-22

|  |  |  |
| --- | --- | --- |
| Invalid | Valid | Invalid |
| 19-11-22 | 20-11-22, 21-11-22, 29-11-  22, 30-11-22 | 1-12-22 |

Test Case 1 = 19-11-22 (invalid)

Test Case 2 = 20-11-22 (valid)

Test Case 3 = 21-11-22 (valid)

Test Case 4= 29-11-22 (valid)

Test Case 5 = 30-11-22 (valid)

Test Case 6 = 1-12-22 (invalid)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | | Date check | | | **Test Case ID** | | Date check-1A | |
| **Test Case Description** | |  | | | **Test Priority** | | High | |
| **Pre-Requisite** | | NA | | | **Post-requisite** | | NA | |
| Test Execution Steps: | | | | | | | | |
| **S.No** | **Action** | | **Inputs** | **Expected Output** | | **Actual Output** | | **Test Result** |
| 1 | Date Check | | 21-11-22 | Date Valid | | Date Valid | | Pass |
| 2 | Date Check | | 19-11-22 | Date Invalid | | Date Invalid | | Fail |

1. **Media file size range**

If only media files of size between 50kb and 1mb should be uploaded.

|  |  |  |
| --- | --- | --- |
| Invalid | Valid | Invalid |
| 49KB | 50KB, 51KB, 0.99MB, 1MB | 1.1MB |

Test Case 1 = 49kb (invalid) Test Case 2 = 50kb (valid) Test Case 3 = 51kb (valid) Test Case 4= 0.99mb (valid) Test Case 5 = 1mb (valid) Test Case 6 = 1.1mb (invalid)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | | Size check | | | **Test Case ID** | | Date check-1A | |
| **Test Case Description** | |  | | | **Test Priority** | | High | |
| **Pre-Requisite** | | NA | | | **Post-requisite** | | NA | |
| Test Execution Steps: | | | | | | | | |
| **S.No** | **Action** | | **Inputs** | **Expected Output** | | **Actual Output** | | **Test Result** |
| 1 | File Size Check | | 51KB | Valid | | Valid | | Pass |
| 2 | File Size Check | | 1.2MB | Invalid | | Invalid | | Fail |

1. **Weight limit range**

If there is health survey form for particular set of people and there is a input column for weight and condition is there for a valid range (e.g 40 kg - 85kg).

|  |  |  |
| --- | --- | --- |
| Invalid | Valid | Invalid |
| 39 | 40, 41, 84, 85 | 86 |

Test Case 1 = 39 (invalid)

Test Case 2 = 40 (valid)

Test Case 3 =41 (valid)

Test Case 4= 84 (valid)

Test Case 5 = 85 (valid)

Test Case 6 = 86 (invalid)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | | Weight check | | | **Test Case ID** | | Weight check-1A | |
| **Test Case Description** | |  | | | **Test Priority** | | High | |
| **Pre-Requisite** | | NA | | | **Post-requisite** | | NA | |
| Test Execution Steps: | | | | | | | | |
| **S.No** | **Action** | | **Inputs** | **Expected Output** | | **Actual Output** | | **Test Result** |
| 1 | Weight Check | | 41KG | Valid | | Valid | | Pass |
| 2 | Weight Check | | 102KG | Invalid | | Invalid | | Fail |

**Conclusion:**

# PRACTICAL: 02

## **Aim:** Create test cases using equivalence partitioning.

**What is Equivalence Class Partitioning?**

Equivalence Class Testing (also known as Equivalence Partitioning) is a black-box testing technique used to reduce the number of test cases while still covering all possible input scenarios. It works by dividing input data into partitions of equivalent data from which test cases can be derived. The idea is that if one value in the partition works correctly, the others will too.

**Why Use ECP?**

* Reduces the number of test cases
* Covers more logic in less time
* Prevents redundant tests
* Makes sure important boundary cases are tested

Steps in Equivalence Class Testing:

* 1. **Identify the input condition(s)** :- Analyze the input field and define its range or expected values.
  2. **Partition the input data into equivalence classes:**
     + Valid Equivalence Classes – Inputs that should be accepted.
     + Invalid Equivalence Classes – Inputs that should be rejected.
  3. **Select representative values** from each equivalence class to form test cases.

**Example Scenario:**

Online Exam Registration System

**Input Condition:**

* Age of a candidate required to register for a professional certification exam.
* The exam is only available to candidates with ages between **18 and 65 years** (inclusive).
* Age must be a valid integer.

1. **Set Condition**

* **Input:** Age (in years)
* **Rules:** Integer value

1. **Define Valid/Invalid Equivalence Classes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Type** | **Condition** | **Description** | **Example Values** |
| Invalid | Age < 18 | Too young (underage) | 10, 15, 17 |
| Valid | 18 ≤ Age ≤ 65 | Eligible age range | 18, 35, 65 |
| Invalid | Age > 65 | Too old (retirement age) | 66, 70, 80 |
| Invalid | Non-integer | Invalid data type | "twenty", 25.5, null |

1. **Design Test Cases**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Age** | **Expected Result** | **Equivalence Class** |
| TC01 | 15 | Reject | Invalid (Age < 18) |
| TC02 | 18 | Accept | Valid (Boundary - lower) |
| TC03 | 42 | Accept | Valid (Middle range) |
| TC04 | 65 | Accept | Valid (Boundary - upper) |
| TC05 | 70 | Reject | Invalid (Age > 65) |
| TC06 | "twenty-five" | Reject | Invalid (Non-integer string) |
| TC07 | 30.5 | Reject | Invalid (Decimal number) |
| TC08 | null | Reject | Invalid (Null input) |
| TC09 | -5 | Reject | Invalid (Negative number) |
| TC10 | 0 | Reject | Invalid (Zero/infant) |

**Additional Example Scenario:**

University Course Enrollment System

**Input Condition:**

* Credit hours required for a student to enroll in advanced courses.
* Students must have completed between **60 and 150 credit hours** (inclusive) to be eligible.
* Credit hours must be a valid integer.

1. **Set Condition**

* **Input:** Completed Credit Hours
* **Rules:** Integer value

1. **Define Valid/Invalid Equivalence Classes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Type** | **Condition** | **Description** | **Example Values** |
| Invalid | Credits < 60 | Insufficient credits (underclassman) | 30, 45, 59 |
| Valid | 60 ≤ Credits ≤ 150 | Eligible credit range | 60, 90, 150 |
| Invalid | Credits > 150 | Excessive credits (over-enrollment) | 151, 180, 200 |
| Invalid | Non-integer | Invalid input format | "sixty", 75.25, undefined |

1. **Design Test Cases**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Credits** | **Expected Result** | **Equivalence Class** |
| TC01 | 45 | Reject | Invalid (Credits < 60) |
| TC02 | 60 | Accept | Valid (Boundary - lower) |
| TC03 | 105 | Accept | Valid (Middle range) |
| TC04 | 150 | Accept | Valid (Boundary - upper) |
| TC05 | 175 | Reject | Invalid (Credits > 150) |
| TC06 | "ninety" | Reject | Invalid (Non-integer string) |
| TC07 | 82.75 | Reject | Invalid (Decimal number) |

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Input Credits** | **Expected Result** | **Equivalence Class** |
| TC08 | null | Reject | Invalid (Null input) |
| TC09 | -10 | Reject | Invalid (Negative number) |
| TC10 | 0 | Reject | Invalid (No credits completed) |

**Conclusion:**



# PRACTICAL: 03

## **Aim:** Design independent paths by calculating cyclomatic complexity using date problem.

**What is Cyclomatic Complexity?**

The cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program.

**Formula for Calculating Cyclomatic Complexity M = E - N + P**

Where,

E = Number of Edges N = Number of Node

P = Number of Connected Component

**Data problem Code: Example 1**

#include <stdio.h> int main(){

int age;

printf(“Enter the Age:”); scanf(“%d”, &age);

if (age < 18) {

printf(“You are not Eligible”);

}

else if (age > 18) {

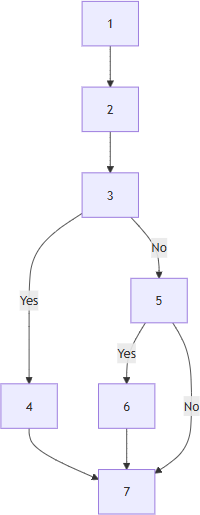
printf(“You are Eligible”);

}

return 0;

}

**Control Flow Graph:**

****

**Calculations**:

Method 1: The graph has 8 nodes and 9 edges. V(G) = 7 - 8 + 2 = 3.

Method 2: There are two predicate nodes (the if age < 18 and else if age > 18 conditions). V(G) = 2 + 1 = 3.

Interpretation: The cyclomatic complexity of this function is 3, meaning there are three independent paths through the code:

1. age < 18 is true  Print(Not Eligible)
2. age > 18 is true  Print(Eligible)
3. x == 18 is true  Control Goes to END

Identification of independent paths(Basic Path set) from the control flow graph:

1) 1-2-3-5-6-7

2) 1-2-3-4-7

3) 1-2-3-5-7



**Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input (age)** | **Expected Output** | **Actual Output** | **Test Result** |
| TC1 | 15 | You are not Eligible | As expected | Pass |
| TC2 | 20 | You are Eligible | As expected | Pass |
| TC3 | 18 | (No Output) | As expected | Pass |

**Data problem Code: Example 2**

#include <stdio.h> int main() {

int a, b;

printf("Enter two numbers: "); scanf("%d %d", &a, &b);

if (a > 0) {

printf("A is positive\n");

} else if (a < 0) {

printf("A is negative\n");

} else {

printf("A is zero\n");

}

if (b % 2 == 0) {

printf("B is even\n");

} else {

printf("B is odd\n");

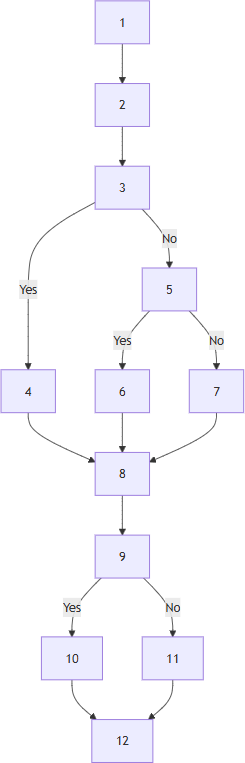
}

return 0;

}



**Control Flow Graph:**

****

**Calculations**:

Method 1: The graph has 8 nodes and 9 edges. V(G) = 14 - 12 + 2 = 4.

Method 2: There are two predicate nodes (the if (a > 0), else if (a < 0) and if (b % 2 == 0) conditions).

V(G) = 2 + 1 = 3.

Interpretation: The cyclomatic complexity of this function is 3, meaning there are three independent paths through the code:

1. a > 0 is true, b % 2 == 0 is true
2. a > 0 is true, b % 2 != 0 is true
3. a < 0 is true, b % 2 == 0
4. a < 0 is true, b % 2 != 0
5. a == 0, b % 2 == 0
6. a == 0, b % 2 != 0

Identification of independent paths(Basic Path set) from the control flow graph:

1. 1-2-3-4-8-9-10-12

2. 1-2-3-4-8-9-11-12



3. 1-2-3-5-6-8-9-10-12

4. 1-2-3-5-6-8-9-11-12

5. 1-2-3-5-7-8-9-10-12

6. 1-2-3-5-7-8-9-11-12

**Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input (a, b)** | **Expected Output** | **Actual Output** | **Test Result** |
| TC1 | 5, 4 | A is positive, B is even | As expected | Pass |
| TC2 | 5, 3 | A is positive, B is odd | As expected | Pass |
| TC3 | -2, 8 | A is negative, B is even | As expected | Pass |
| TC4 | -2, 5 | A is negative, B is odd | As expected | Pass |
| TC5 | 0, 6 | A is zero, B is even | As expected | Pass |
| TC6 | 0, 7 | A is zero, B is odd | As expected | Pass |

**Conclusion:**

# PRACTICAL: 04

**Aim:** Design test cases using a Decision Table and identify independent paths using the DD

Path for the date validation problem.

# Theory:

**What is DD Path Testing?**  
 - DD Path (Decision-to-Decision path) Testing focuses on control flow between decision nodes.  
 - In a Control Flow Graph (CFG), sequences of non-branching nodes are grouped into a single DD-path

node.  
 - This testing ensures branch coverage (C2) and helps in identifying independent execution paths.

# Control Flow Graph Reduction (DD Path Nodes):

|  |  |
| --- | --- |
| Line Numbers | Node |
| 1–6 | a |
| 7 | b |
| 8–9 | c |
| 10 | d |
| 11–12 | e |
| 13–15 | f |
| 16 | g |
| 17 | h |
| 18–19 | i |
| 20 | j |
| 21–22 | k |
| 23–25 | l |
| 26 | m |
| 27–29 | n |
| 30 | o |

# Independent Paths Identified from the DD Path Graph:

Here are the 6 independent paths required to achieve complete branch coverage:

1) a-b-c-d-e-g-m-o

2) a-b-c-d-f-g-m-o

3) a-b-h-i-m-o

4) a-b-h-j-k-m-o

5) a-b-h-j-l-m-o

6) a-n-o

# Decision Table for Date Validation:

Conditions:

C1: Month is 31-day month (1,3,5,7,8,10,12)

C2: Month is 30-day month (4,6,9,11)

C3: Month is February

C4: Day ≤ 31

C5: Day ≤ 30

C6: Day ≤ 29

C7: Year is leap year

C8: Day ≤ 28

Actions:

A1: Valid Date

A2: Invalid Date

Decision Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rule | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | Action |
| R1 | Y | - | - | Y | - | - | - | - | A1 |
| R2 | Y | - | - | N | - | - | - | - | A2 |
| R3 | - | Y | - | - | Y | - | - | - | A1 |
| R4 | - | Y | - | - | N | - | - | - | A2 |
| R5 | - | - | Y | - | - | Y | Y | - | A1 |
| R6 | - | - | Y | - | - | Y | N | - | A2 |
| R7 | - | - | Y | - | - | N | - | Y | A1 |
| R8 | - | - | Y | - | - | N | - | N | A2 |

# 

# Conclusion: