

Lecture 08 : Software Testing

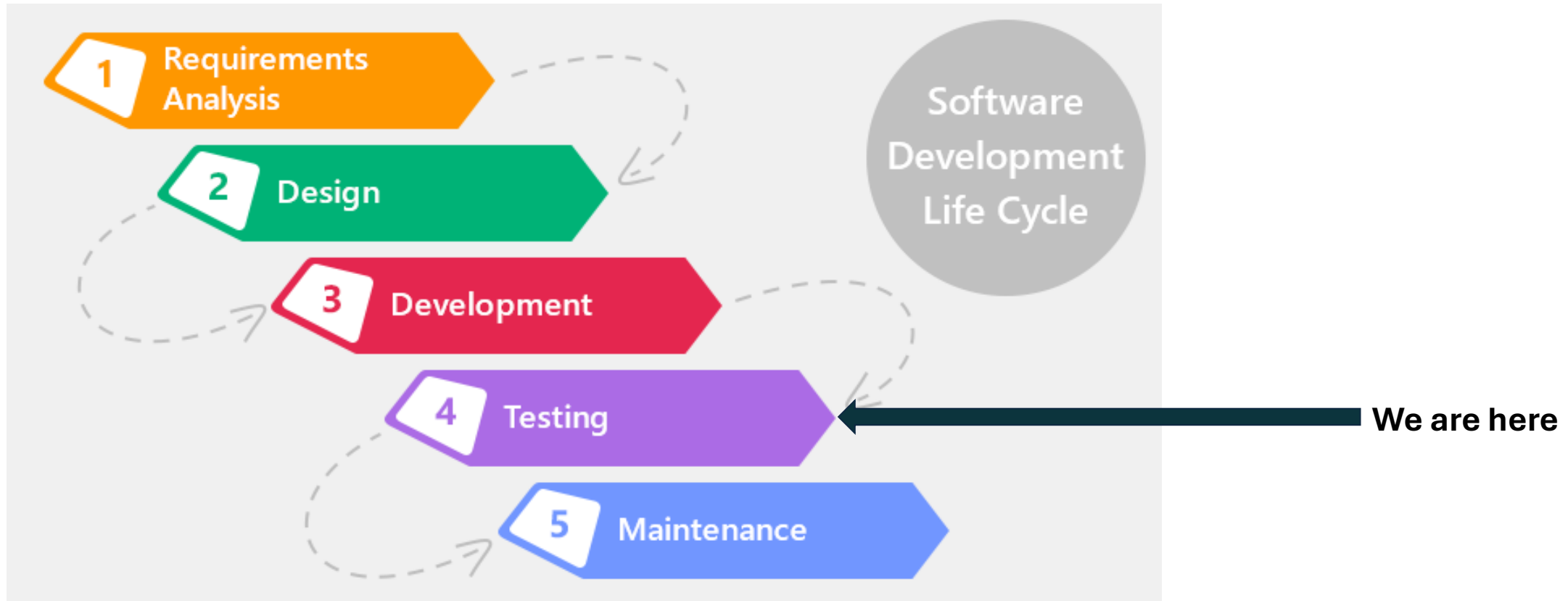
SE2030 – Software Engineering

Lesson Learning Outcomes

By the end of this lecture, students will be able to:

- Explain the importance of software testing in ensuring software quality and user satisfaction.
- Distinguish between verification and validation in the testing process.
- Apply test design techniques such as Equivalence Partitioning (EP) and Boundary Value Analysis (BVA).
- Analyse code quality using white-box testing methods like statement and branch coverage.
- Compare functional and non-functional testing types.

Introduction to Software Testing



Introduction to Software Testing

Why Software Testing Matters?

Opening Scenario:

Imagine you're booking an airline ticket. You pay, get a confirmation, but at the airport, they say no ticket exists in your name. You paid, but you can't board.

Would you use this airline's website again?



Introduction to Software Testing



Famous Software Failure

Heathrow International Airport Terminal 5 baggage handling system failure (2008, UK)

Cause: System not tested under real operational load

Impact:

- 40,000 bags lost or delayed
- 500 flights cancelled
- £16 million in costs
- Significant reputational damage for British Airways

Introduction to Software Testing

What is Software Testing?

Software Testing is the process of **evaluating a software application** to check whether it meets:

- **Specified requirements**

(Does it do what the client asked for?)

- **End-user needs**

(Is it useful, usable, and reliable for real users?)

Purpose:

- To ensure the software behaves **as intended**.
- To **detect defects** before customers encounter them.
- To improve **reliability, safety, and user satisfaction**.



Introduction to Software Testing

Definition

“Software Testing is the process of executing a program or system with the intent of finding errors” [Myers, 79]

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence” [Dijkstra, 1972]

Introduction to Software Testing

Examples

- **Mobile Banking Application**

The application processes transfers correctly, but the interface is so confusing that users accidentally send money to the wrong account.

Why testing matters: Meeting end-user needs involves ensuring the system is both user-friendly and functionally correct.

- **Online Exam System**

Students take an online exam, but after submitting their answers, they disappear due to a server error.

Why testing matters: Users expect their answers to be safely recorded; failing to do so breaks trust.

Introduction to Software Testing

Example video on software testing:

<https://www.youtube.com/watch?v=TDynSmrzpXw>

Verification Vs Validation

Verification



Did we build the chair according to blueprint?

Validation



Is the chair comfortable to sit on?

- **Verification:**

"Are we building the product, right?"

- The software should conform to its specifications, including both functional and non-functional requirements.

- **Validation:**

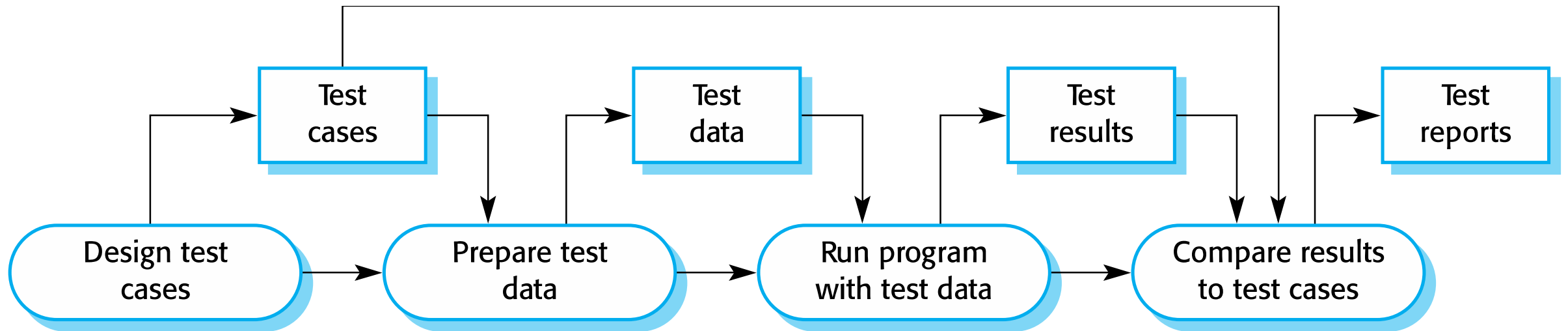
"Are we building the right product?"

- The software should fulfil the user's actual needs, even if they differ from the documented specification.

Verification Vs Validation

Verification	Validation
1. It is the process to ensure whether we are developing the product right or not. (Whether we are developing it according to the plans and specifications)	1. It is the process to test/check whether the product we developed is correct. Whether it satisfies the client requirements.
2. It is a static method of checking the documents, designs, program code, database schemas and specifications.	2. Is a dynamic method of testing the real product.
3. Inspections, reviews and walkthroughs	3. Testing frameworks , testing strategies are used.
4. Low – level activity	4. High – level activity
5. It does not involve code execution	5. It involves code execution
6. Low cost compared to validation tests	6. Costly compared to verification tests

Testing Process

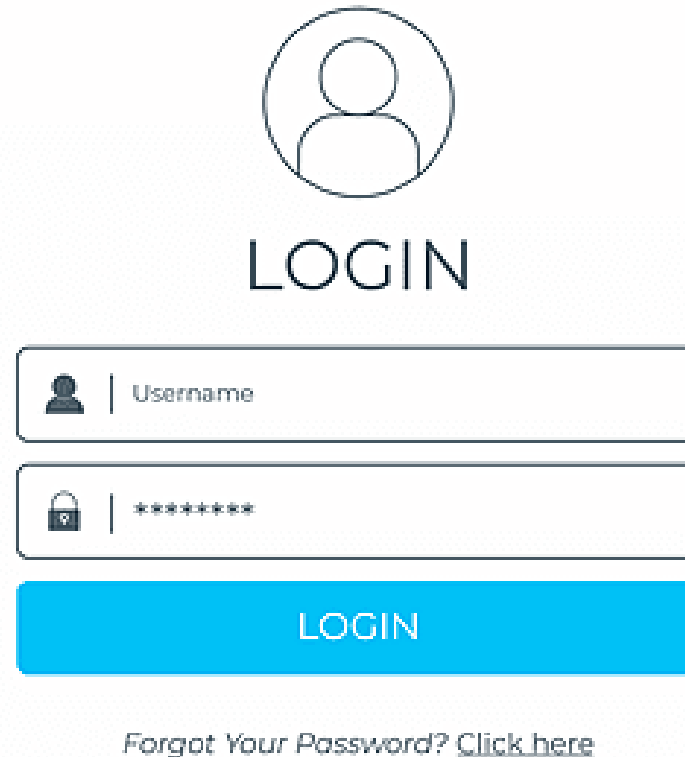


Test Case Designing

Sample Test Case Document

Test Case ID	Test Title	Description	Preconditions	Test Steps	Test Data	Expected Output	Actual Output	Status

Test Case Designing



A login interface mockup. At the top is a circular icon containing a stylized person. Below the icon is the word "LOGIN" in a large, black, sans-serif font. Underneath are two input fields: the first has a person icon and the placeholder text "Username"; the second has a lock icon and placeholder text "*****". Below these fields is a solid blue button with the word "LOGIN" in white. At the bottom, there is a link that reads "Forgot Your Password? [Click here](#)".

Example: *Testing Login UI*

- Enter correct username/password → Login success → Redirect to the home page →  **Pass**
- Enter wrong password → Login success →  **Fail**

Test Case Designing

Test Case ID	Test Title	Description	Preconditions	Test Steps	Test Data	Expected Output	Actual Output	Status
TC_1	Verify login with valid credentials	Ensure that a user can successfully log in using a valid username and valid password	1. User must be registered 2. User must be on the login page	1. Open the login page 2. Enter valid username 3. Enter valid password 4. Click the "Login" button	Username: user1 Password: Pw@123	User is successfully redirected to the homepage	(To be filled after execution)	(To be filled after execution)

Test Case Designing

Test Case ID	Test Title	Description	Preconditions	Test Steps	Test Data	Expected Output	Actual Output	Status
TC_1	Verify login with valid credentials	Ensure that a user can successfully log in using a valid username and valid password	1. User must be registered 2. User must be on the login page	1. Open the login page 2. Enter valid username 3. Enter valid password 4. Click the "Login" button	Username: user1 Password: Pw@123	User is successfully redirected to the homepage	User is successfully redirected to the homepage	Pass

Test Design Techniques

1. Black-Box Testing (Functional Testing)

Focused on the *system's functionality without examining its* internal code.

2. White-Box Testing (Structural Testing)

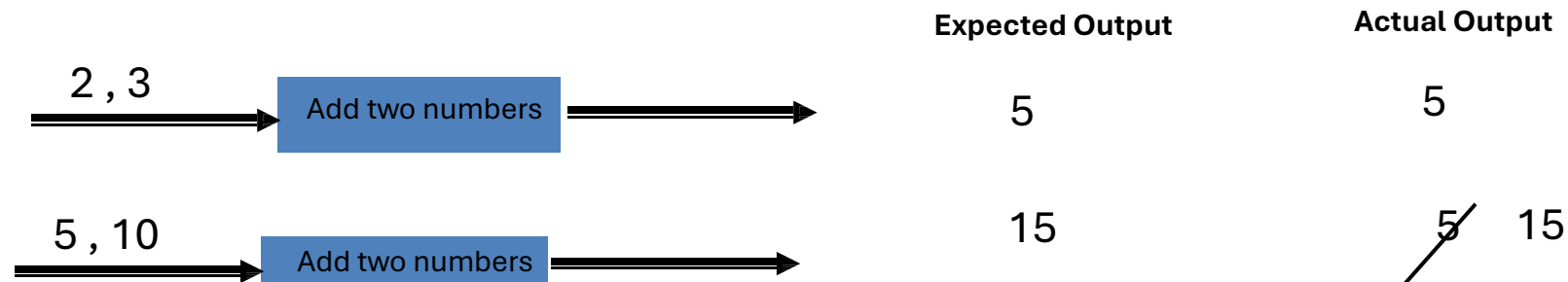
Focused on the internal structure of the code.

Black-Box Testing

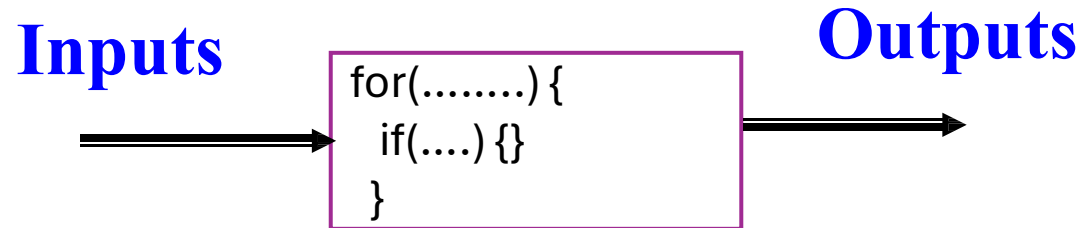
Testing focuses on the software functional requirements and input/output.



Module under test is treated as a Black Box



White-Box Testing



Testing is based on the structure of the program

- In white box testing internal structure of the program is taken into account.
- The test data is derived from the structure of the software.
- Should have programming knowledge.

Black-Box Testing Vs White-Box Testing

Black Box Testing	White Box Testing
The Internal Working of an application are not required to be known	Tester has full knowledge of the Internal workings of the application
Also known as closed box testing , data driven testing and functional testing	Also known as clear box testing , structural testing or code-based testing
Performed by end users and also by testers and developers	Normally done by testers and developers
This is the least time consuming and least exhaustive	The most exhaustive and time-consuming type of testing

Black-Box Testing Strategies

1. Equivalence Partitioning

Focused on dividing the input data into groups (partitions) where test cases from one group are expected to behave the same.

2. Boundary Value Analysis

Focused on testing values at the boundaries of input ranges, since errors often occur at the edges.

Equivalence Partitioning

Example Equivalence Partitioning:

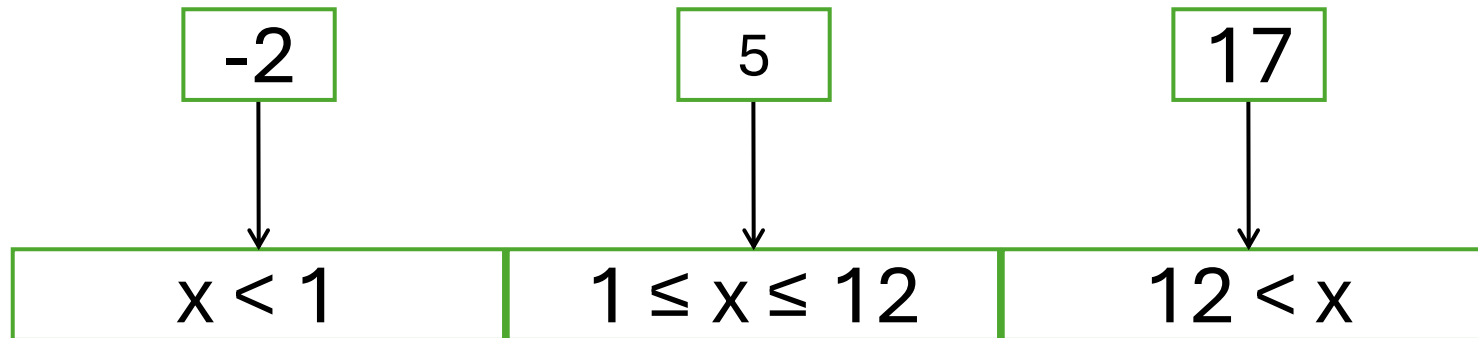
- Example of a function that takes a parameter “month”.
- The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition.
- In this example, there are two further partitions of invalid ranges.

$x < 1$	$1 \leq x \leq 12$	$12 < x$
---------	--------------------	----------

Equivalence Partitioning

Example Equivalence Partitioning:

- Test cases are chosen so that each partition will be tested.



Equivalence Partitioning

- In equivalence partitioning, only one condition from each partition needs to be tested.
- This is based on the assumption that all conditions within a partition will be handled in the same way by the application.
- Equivalence partitioning is effective because it reduces the number of tests required.

Activity

In a Library, students can borrow books. There is a limit given for students on the number of books they can borrow at one time. A student can borrow 2-5 books at one time.

With equivalence partitioning, identify the test values to check the borrowing **limit** of a student at **one time**.

Boundary Value Analysis

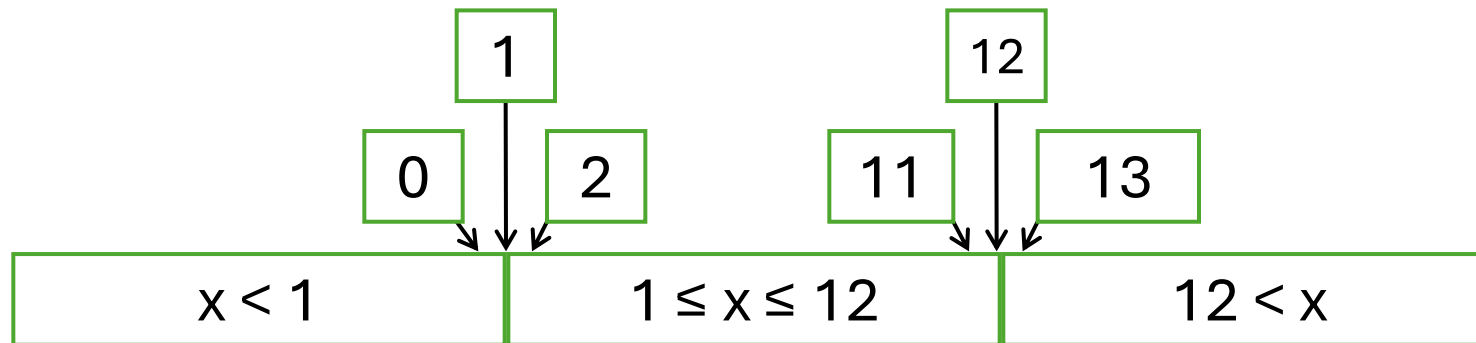
Equivalence Partitioning is usually combined with Boundary Value Analysis; it is not used alone for test case design.

Boundary Value Analysis tests values at the edges of partitions and the smallest values just inside and outside those edges.

Boundary Value Analysis

Example: Boundary Value Analysis

- Example of a function that takes a parameter “month”.
- The valid range for the month is 1 to 12, representing January to December.
- In this example, there are two further partitions of invalid ranges.
- Test cases are supplemented with **boundary values**.



Activity

Write test cases for an input box accepting numbers between 1 and 1000 using Boundary value analysis

Activity

A text field accepts numbers from 1 to 100. Using Equivalence Partitioning (EP) and Boundary Value Analysis (BVA), identify the test cases.

White-Box Testing Strategies

1. Statement Coverage

Execute all statements at least once

2. Branch (Decision/Edge) Coverage

Execute each decision direction at least once

3. Condition (Predicate) Coverage

Execute each decision with all possible outcomes at least once

4. Decision/Condition Coverage

Execute all possible combinations of condition outcomes in each decision

5. Multiple Condition Coverage

Invoke each point of entry at least once Execute all statements at least once

Statement Coverage

Statement Coverage in Testing refers to a white-box testing technique that measures the **percentage of executable statements in the source code that have been executed at least once during testing.**

The minimum number of test cases we need to execute **all the statements** in the program at **least once.**

$$\text{Statement Coverage} = \frac{\text{No of executed statements}}{\text{Total no of statements}} * 100\%$$

Example

Calculate the no of minimum test cases needed for full statement coverage for the given scenario.

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```


Example

Step 1: What is the total number of statements in the code?

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

Diagram illustrating the counting of statements in the code:

- 1: Printsum (int a, int b) {
- 2: int result = a+ b;
- 3: If (result> 0)
- 4: Print ("Positive", result);
- 5: Else
- 6: Print ("Negative", result);
- 7: }

Total no of statements = 7

Example

Step 2: Find out the executed no of statements for Test case 1

Test Case 1 – if a=3, b=9

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}  
}
```

1	✓
2	✓
3	✓
4	✓
5	✗
6	✗
7	✓

**No of executed
statements = 5**

Example

Step 3: Calculate the Statement Coverage

Test Case 1 – if a=3, b=9

```
Printsum (int a, int b) { int result
    = a+ b; If (result> 0)
        Print ("Positive", result);
    Else
        Print ("Negative", result);
}
```

No of executed statements = 5

Total no of statements = 7

Statement coverage = $5/7 * 100 = 71\%$

Example

Step 4: Again, check the statement coverage for Test case 2

Test Case 2 – if a= -2, b= -3

Printsum (int a, int b) {

 int result = a+ b;

 If (result> 0)

 Print ("Positive", result);

 Else

 Print ("Negative", result);

}



Example

Step 5: Again, check the statement coverage for Test case 2

Test Case 2 – if a=-2, b=-3

```
Printsum (int a, int b) {  
    int result = a+ b; If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

No of executed statements = 6
Total no of statements = 7
Statement coverage = $6/7 * 100 = 85\%$

Overall, all the statements are fully covered by using the two test cases. Therefore, the overall statement coverage of 100% can be achieved by the above two test cases.

Activity

Calculate the no of minimum test cases needed for full statement coverage for the given scenario.

```
int f1(int x, int y){  
    while (x != y){  
        if (x>y)  
            x=x-y; else  
            y=y-x;  
    }  
  
    return x; }
```

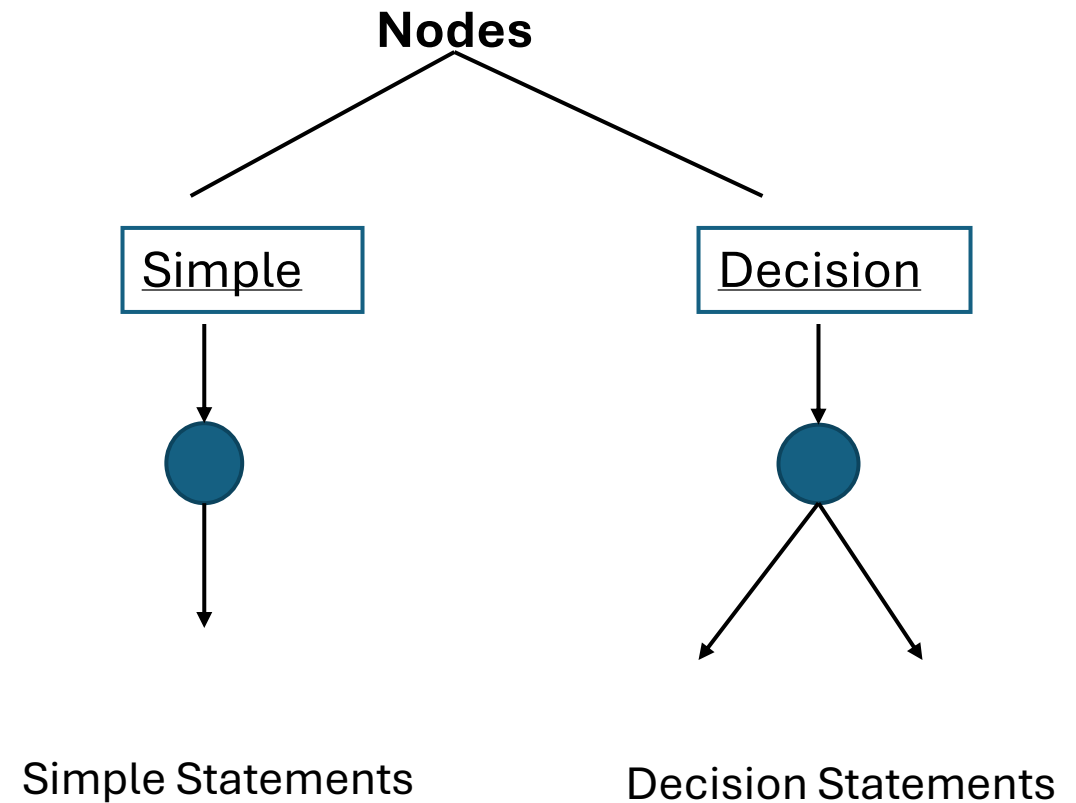
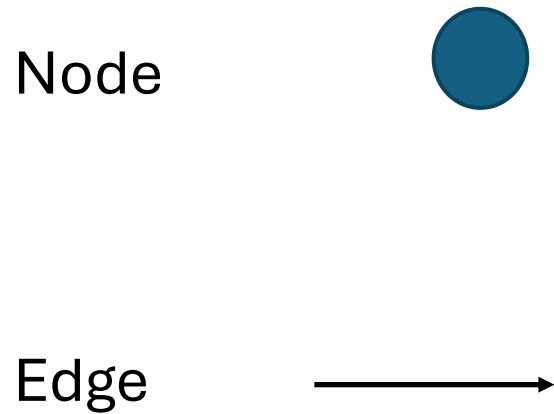
Branch Coverage

Branch Coverage (also called Decision Coverage) is a white-box testing technique that **measures whether each possible branch (true/false decision) of control structures like if, else, switch, loops has been executed at least once.**

The minimum number of test cases we need to execute **all the branches** in the program at **least once**.

$$\text{Branch Coverage} = \frac{\text{No of executed branches}}{\text{Total no of branches}} * 100\%$$

Control Flow Graph



Example

Calculate the no of minimum test cases needed for full branch coverage for the given scenario.

```
Printsum (Int a, Int b) {  
    Int result = a+ b;  
    If (result> 0)  
        Print ("Positive",  
result); else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;
```

Step 1: Come up with a simple control flow graph for the given code.

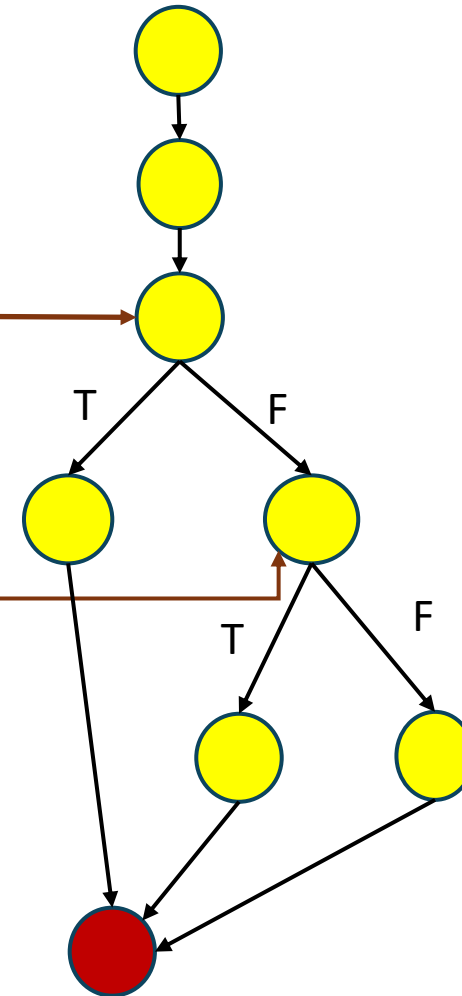
```
Printsum (Int a, Int b){  
    Int result = a+b;
```

```
    if(result>0)  
        Print ("Positive", result);
```

```
    else if (result<0)  
        Print ("Negative", result);
```

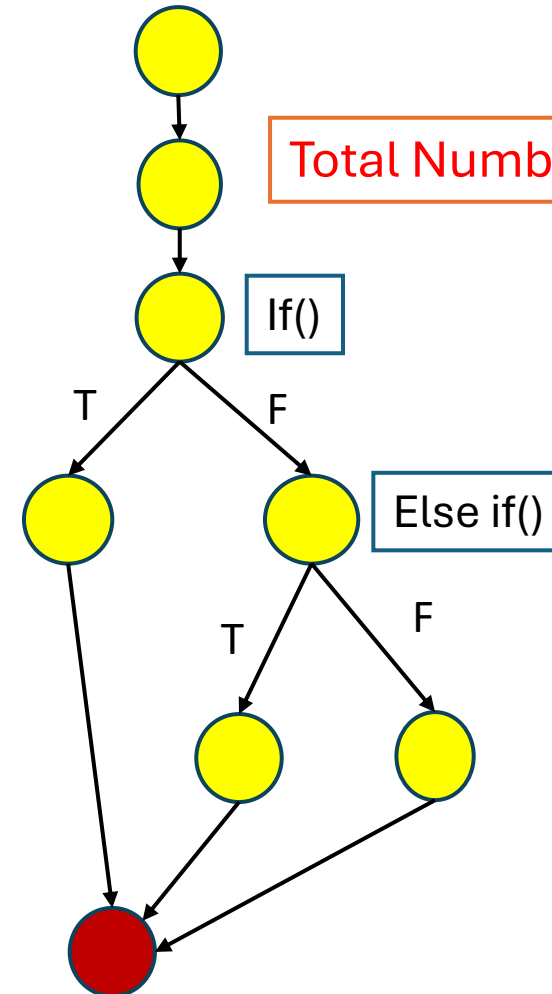
```
    else  
        do nothing;
```

```
}
```



Step 2: Total number of branches in the program.

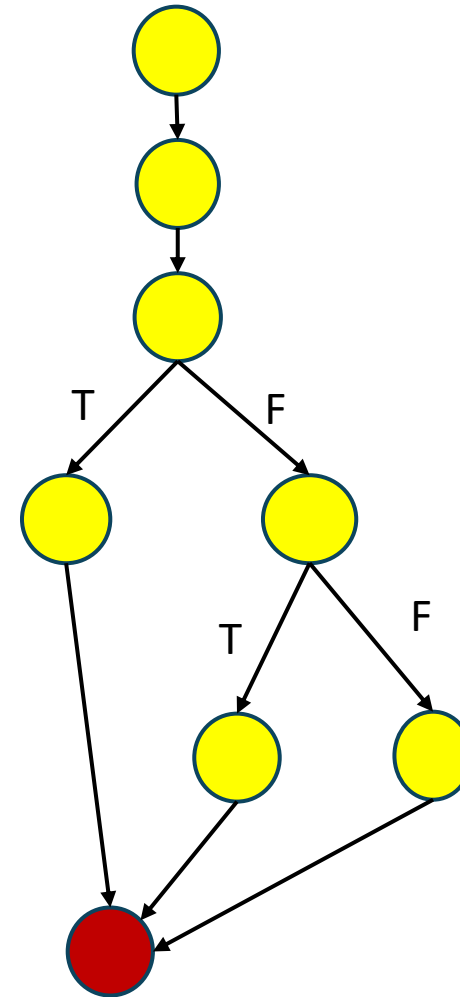
```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```



Step 3: Traverse through the graph when a=3 and b=5

Test case 1 :- a=3,b=5

```
Printsum (Int a, Int b){  
    Int result =  
    a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

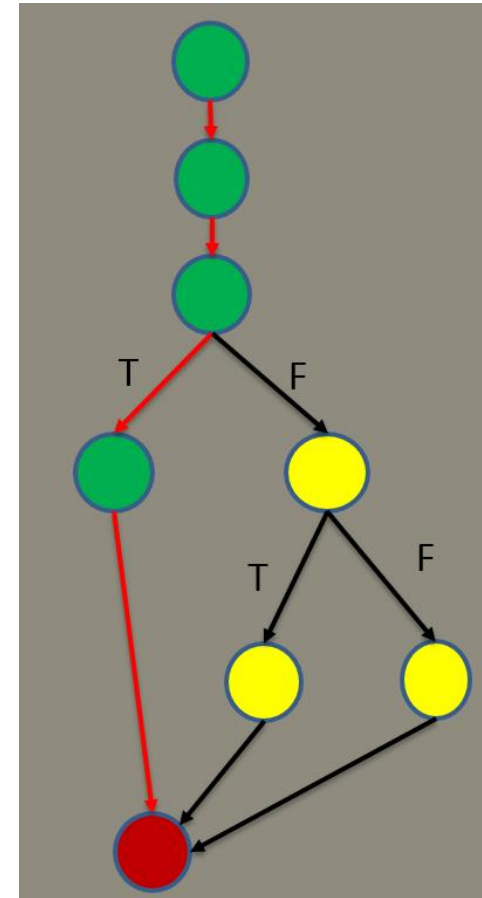
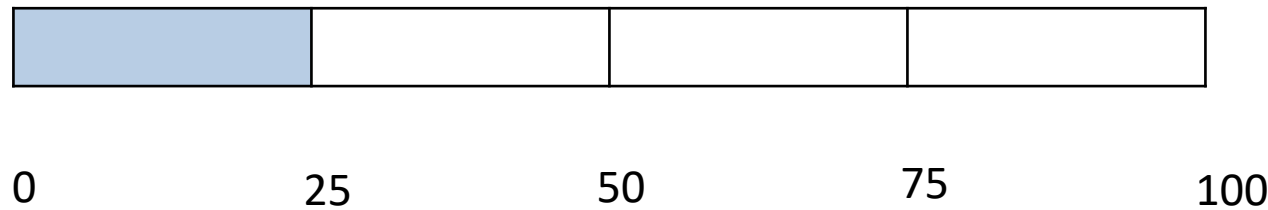


Step 4: Calculate the branch coverage when a=3 and b=5.

Test case 1 :- a=3, b=5

This test case covers the path/branch highlighted.

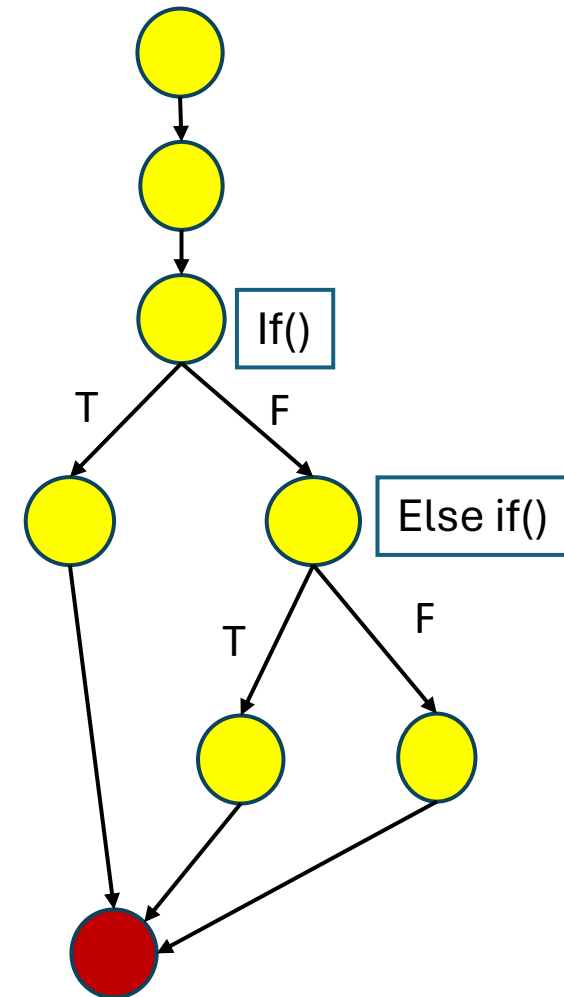
branch coverage = $\frac{1}{4} * 100 = 25\%$



Step 5: Calculate the branch coverage when a=-5 and b=-8.

Test case 2 – a=-5,b=-8

```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

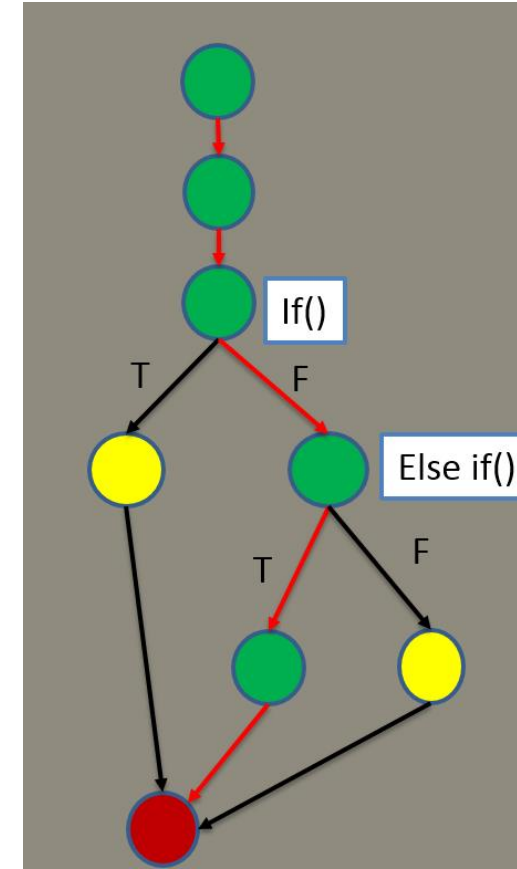
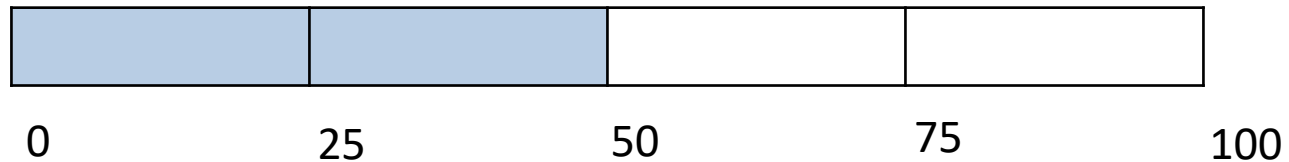


Step 5: Calculate the branch coverage when a=-5 and b=-8.

Test case 2 – a=-5,b=-8

This test case covers the path highlighted. Two branches are covered.

$$\text{Branch coverage} = 2/4 * 100 = 50\%$$

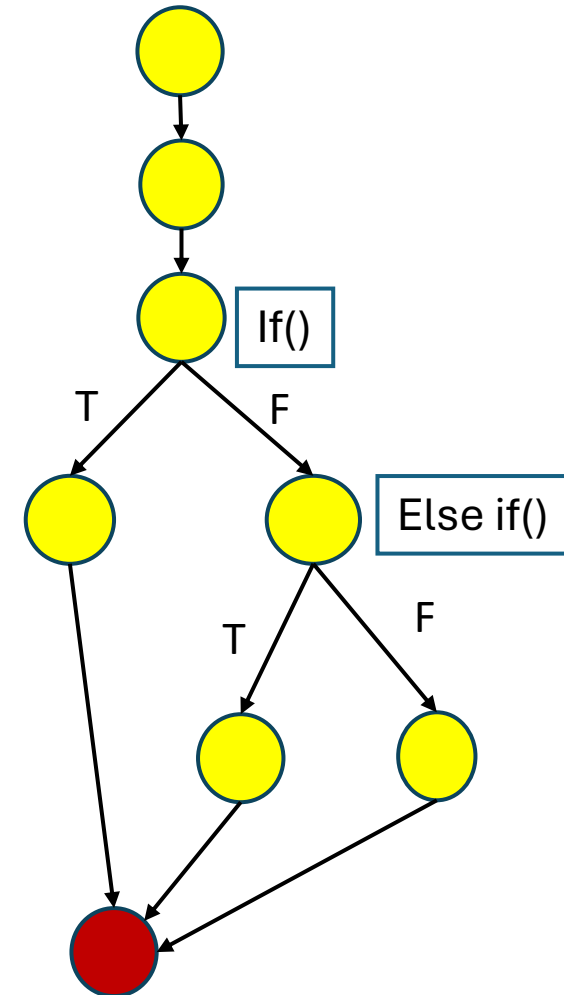


Step 5: Calculate the branch coverage when a and b are 0.

Test case 3:- a=0, b=0

```
Printsum (Int a, Int b){  
    Int result = a+b;
```

```
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

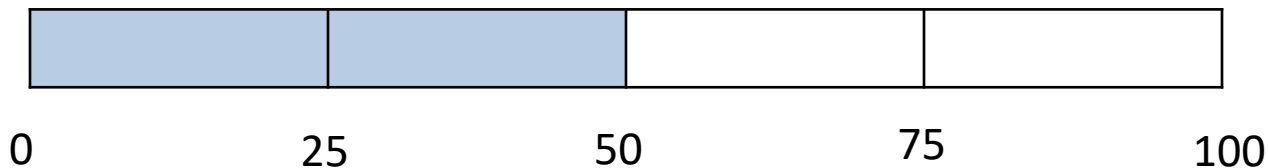


Step 5: Calculate the branch coverage when a and b are 0.

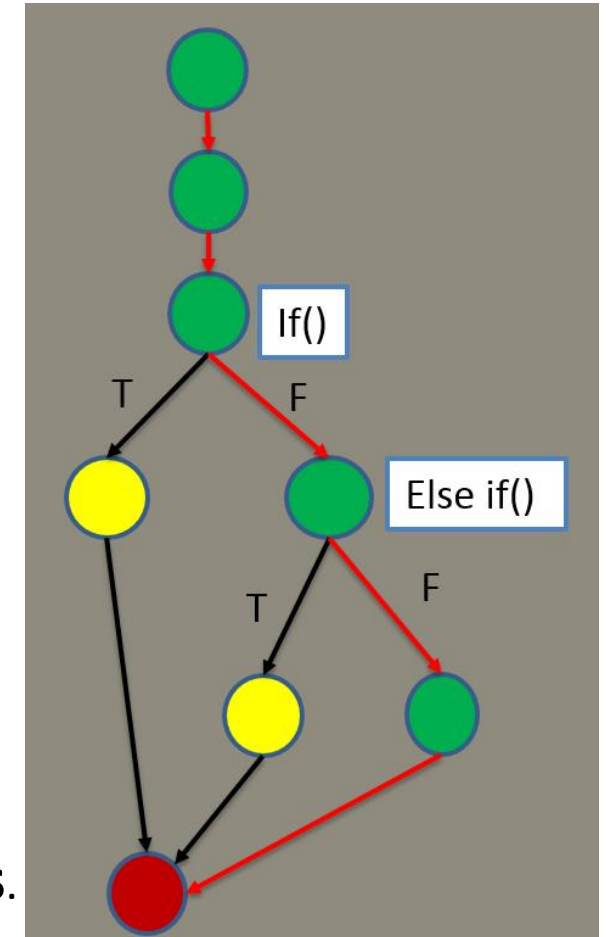
Test case 3 – a=0,b=0

This test case covers the path highlighted.
Two branches are covered.

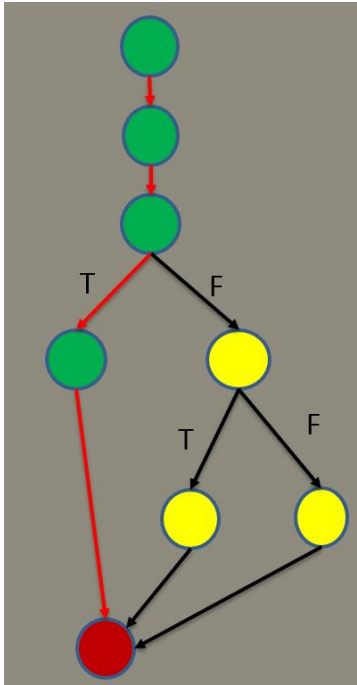
branch coverage = $2/4 * 100 = 50\%$



By using minimum three test cases we can test all the branches.

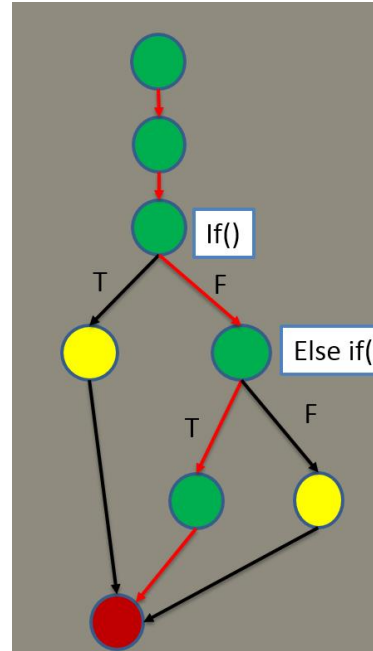


Test case 1 :- a=3, b=5



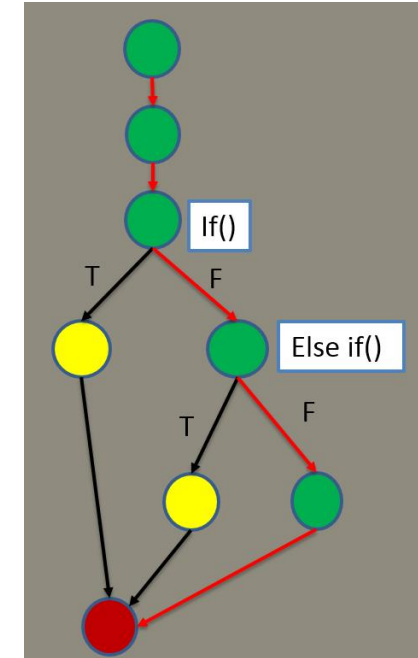
branch coverage= $1/4 \times 100 = 25\%$

Test case 2 :- a=-5, b=-8



branch coverage= $2/4 \times 100 = 50\%$

Test case 3:- a=0, b=0



branch coverage= $2/4 \times 100 = 50\%$

By using minimum three test cases we can test all the branches.

Software Testing Types

1. Functional Testing

Tests the system's functionalities and ensures that it behaves as expected.

2. Non-Functional Testing

Tests the system's non-functional aspects like performance, security, usability, and reliability.

Functional Testing

- **Unit testing:**

Individual program units or object classes are tested.

Unit testing should focus on testing the functionality of objects or methods.

- **Integration testing:**

Several individual units are integrated to create composite components.

Component testing should focus on testing component interfaces.

- **System testing:**

Some or all of the components in a system are integrated, and the system is tested as a whole.

System testing should focus on testing component interactions.

- **Acceptance Testing:**

Customers test a system to decide whether or not it is ready to be accepted by the system developers and deployed in the customer environment. Primarily for custom systems.

Non-Functional Testing

Performance testing:

Ensure that a software program or system meets specific performance goals, such as response time or throughput.

Volume Testing:

Ensure that a software program or system can handle a large volume of data.

For example, if the website is developed to handle traffic of 500 users, volume testing will determine whether the site can handle 500 users or not.

Security testing:

Ensure that a software program or system is secure from unauthorized access or attack.

Non-Functional Testing

Compatibility testing:

Ensure that a software program or system is compatible with other software programs or systems.

Usability Testing:

Ensure that a software program or system is easy to use.

Functional Vs Non-Functional Testing

Feature	Functional Testing	Non-Functional Testing
Definition	Tests the system's functionalities and ensures that it behaves as expected.	Tests the system's non-functional aspects like performance, security, usability, and reliability.
Objective	Ensures that the application works as per the functional requirements.	Ensures that the system meets non-functional requirements like speed, scalability, and user experience.
Focus Areas	Verifies business logic, database operations, APIs, user interfaces, and system workflows.	Checks aspects such as performance, load handling, security, compatibility, and usability.
Testing Techniques	Includes Unit Testing, Integration Testing, System Testing, User Acceptance Testing (UAT).	Includes Performance Testing, Load Testing, Stress Testing, Security Testing, Usability Testing, etc.
Outcome	Determines whether the system meets functional requirements.	Determines how well the system performs under various conditions.

Thank You!