Software Testing

Software Testing: Introduction

- activity to check whether the actual results match the expected results.
- ensure that the software system is defect free.
- helps to identify errors, gaps or missing requirements

Software Testing: Why Important?

- Software bugs could be expensive
- Software bugs could be dangerous
- Software bugs can potentially cause monetary and human loss

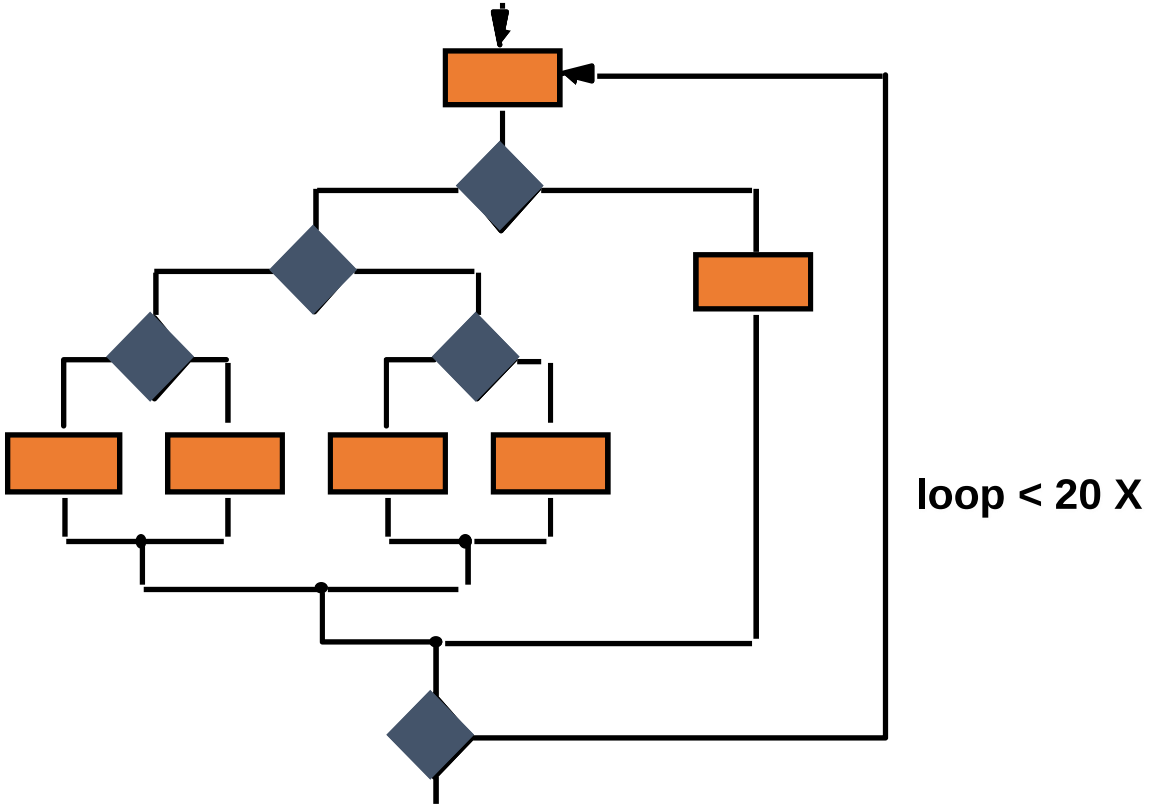
What is a "Good" Test?

- A good test has a high probability of finding an error
- A good test is not redundant
- A good test should be "best of breed"
  - the test that has the **highest likelihood of uncovering a whole class of errors** should be used
- A good test should be neither too simple nor too complex

Exhaustive Testing

- Exhaustive testing is a test approach in which all possible data combinations are used for testing.

If, there are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

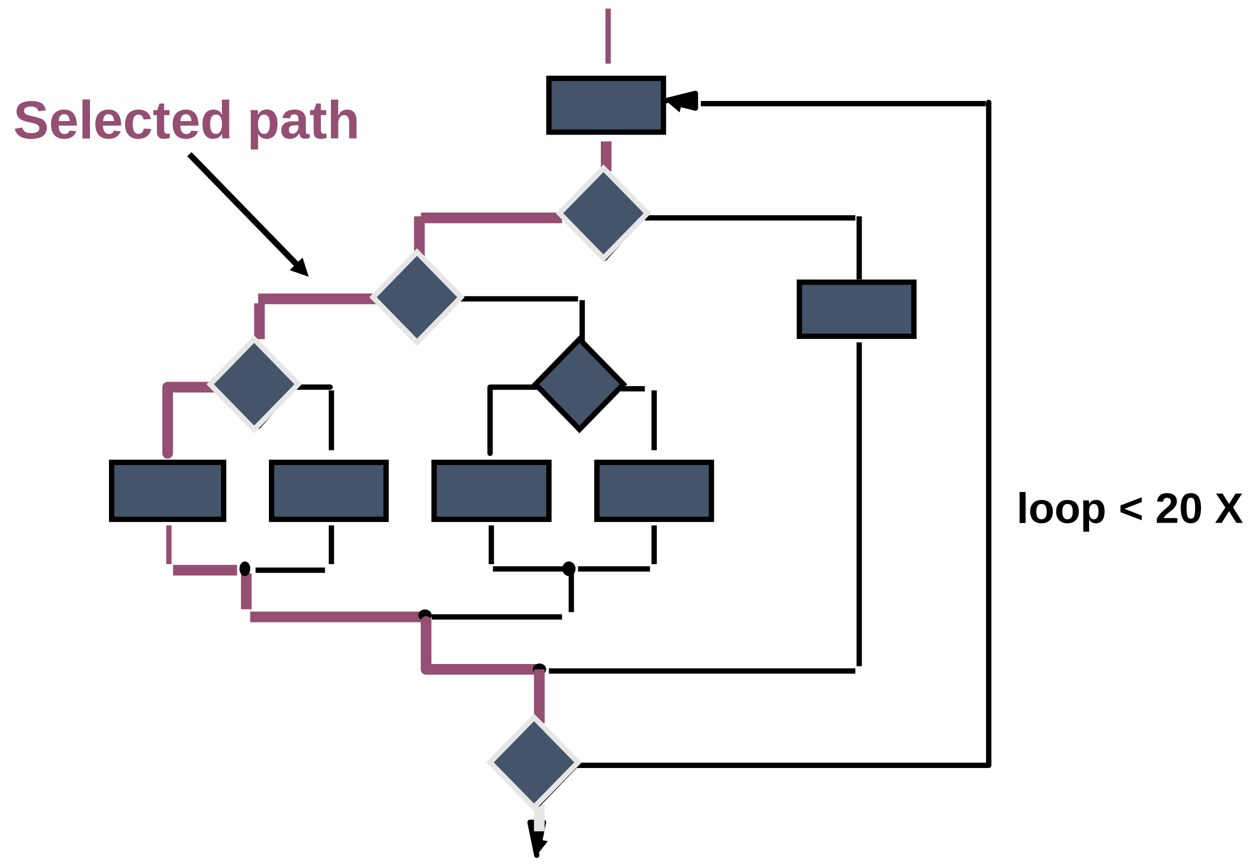


Exhaustive Testing (Example.)

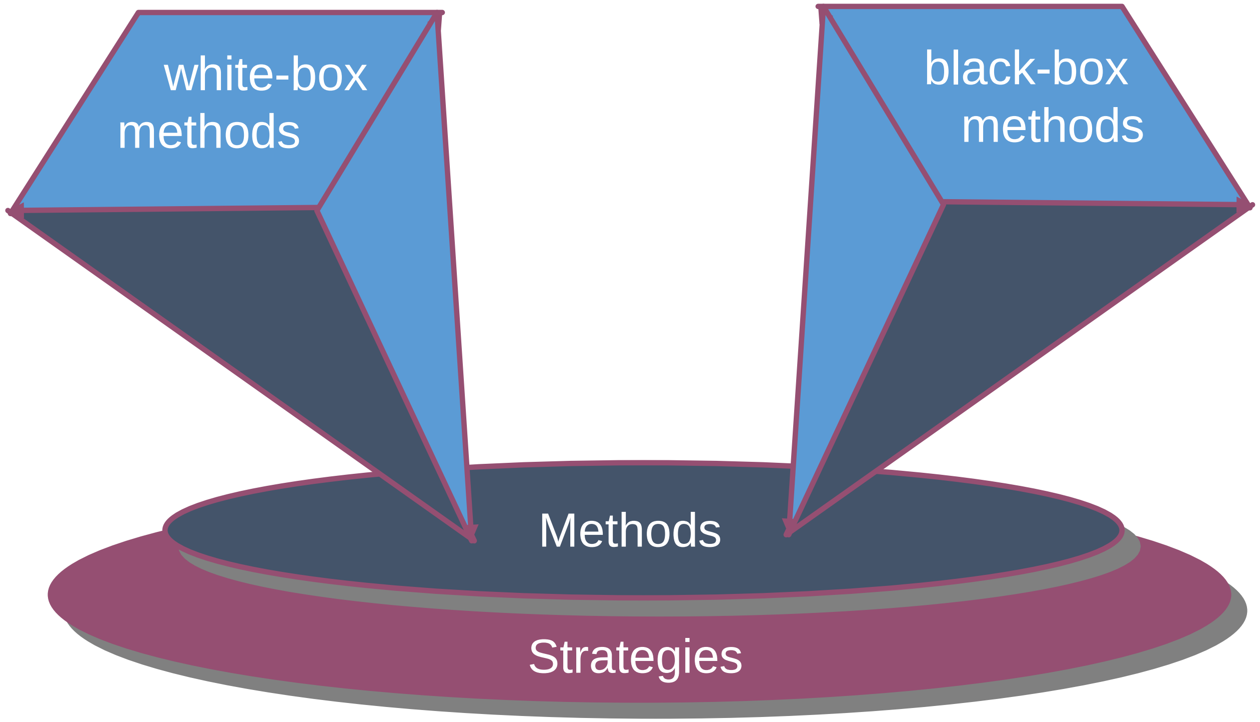
- Consider an application in which a password field
- that accepts 3 characters,
- with no consecutive repeating entries.
- Hence, there are 26 * 26 * 26 input permutations for alphabets (small letter) only. Including special characters and standard characters, there are much more combinations.

Selective Testing

- Only selected path are considered for testing.



Software Testing



How to define Software Testing Principles

Testing:

The execution of a program to find its faults

Verification:

- The process of proving the programs correctness.
- During each development phase

Validation:

- The process of finding errors by executing the program in a real environment
- At the end of development

Debugging:

Diagnosing the error and correct it

Various Types of Software Testing

- Unit Testing (White Box)
- Function Testing (Black Box)
- Integration Testing
- Regression Testing
- System Test
- Acceptance Tests

Black Box Testing

- Also known as Behavioral Testing
- Tester doesn't know the internal structure/design/implementation of the item being tested
- Applicable to:
  - Integration Testing
  - System Testing
  - Acceptance Testing

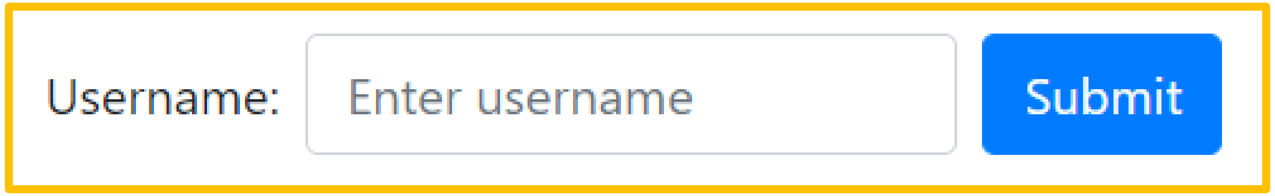
Techniques to Perform Black Box Testing

- **Equivalence Partitioning:**
  - input values into valid and invalid partitions
  - selecting representative values from each partition as test data
- **Boundary Value Analysis:**
  - selecting values that are at the boundaries and just inside/ outside of the boundaries as test data
- **Cause-Effect Graphing:**
  - involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph

Equivalence Partitioning

- Test case design technique
- Divide input data into partitions

Example:
Username field allows 6-10 characters



Invalid Partition	Valid Partition	Invalid Partition
0 to 5 characters	6 to 10 characters	11 or more characters

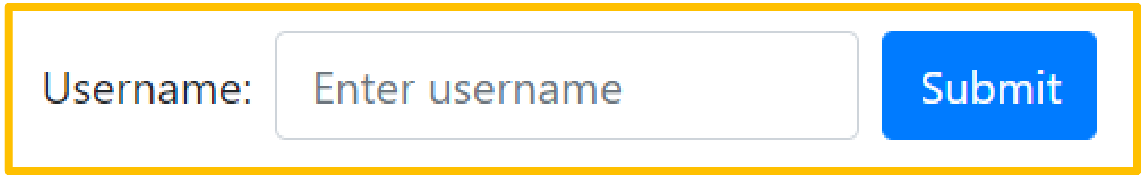
- Pick any value from the invalid partition range – system rejects it
- Pick any value from the valid partition range – system accepts it
- Advantage: Doesn’t require all the values to be checked

Boundary Value Analysis

- Test case design technique
- Testing both sides of each boundary

Example:

Username field allows 6-10 characters



Just Below the Lower Boundary	Near the Lower Boundary	Near the Upper Boundary	Just Above the Upper Boundary
5 characters	6 or 7 characters	9 or 10 characters	11 characters

White Box Testing

- Also known as Clear Box Testing, Open Box Testing, Glass Box Testing
- Tester knows the internal structure/design/implementation of the item being tested
- White box testing is testing beyond the user interface
- Applicable to:
  - Unit Testing
  - System Testing
  - Integration Testing

Error, Fault, Failure

- Error:

  Refers to difference between Actual Output and Expected output.

- Fault:

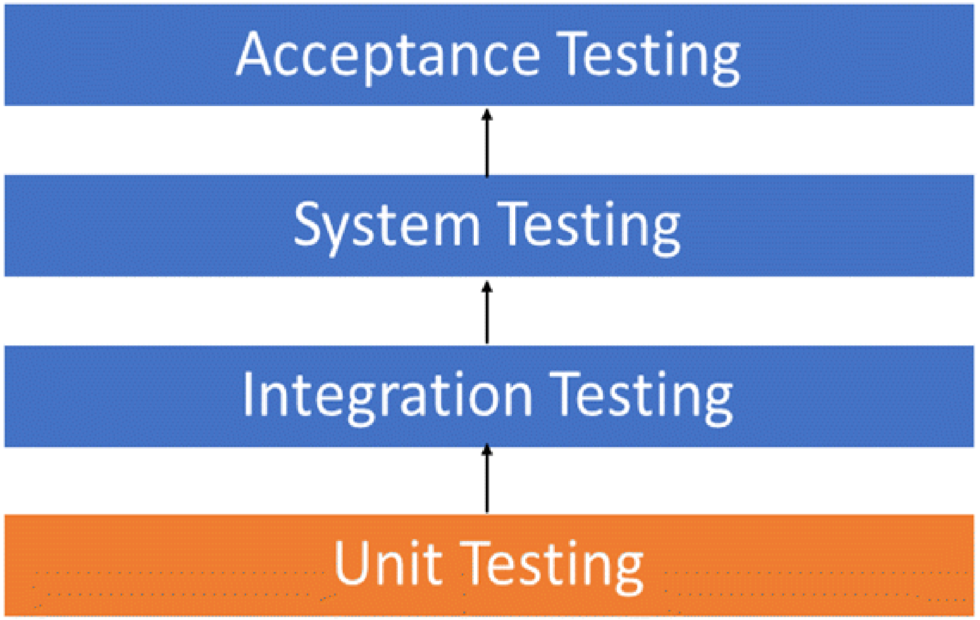
  It is a condition that causes the software to fail to perform its required function.

- Failure:

  Inability of a system or component to perform required function according to its specification.

Testing Levels

Unit Testing ➡ Integration Testing ➡ System Testing ➡ Acceptance Testing



Unit Testing

- White Box Testing method
- First level of software testing
- Normally performed by software developers. In rare cases, it may also be performed by independent software testers.
- Individual units/ components of a software are tested
- Done during the development (coding) of an application
- Proper unit testing done during the development stage saves both time and money in the end
- Unit tests help with code re-use.

Integration Testing

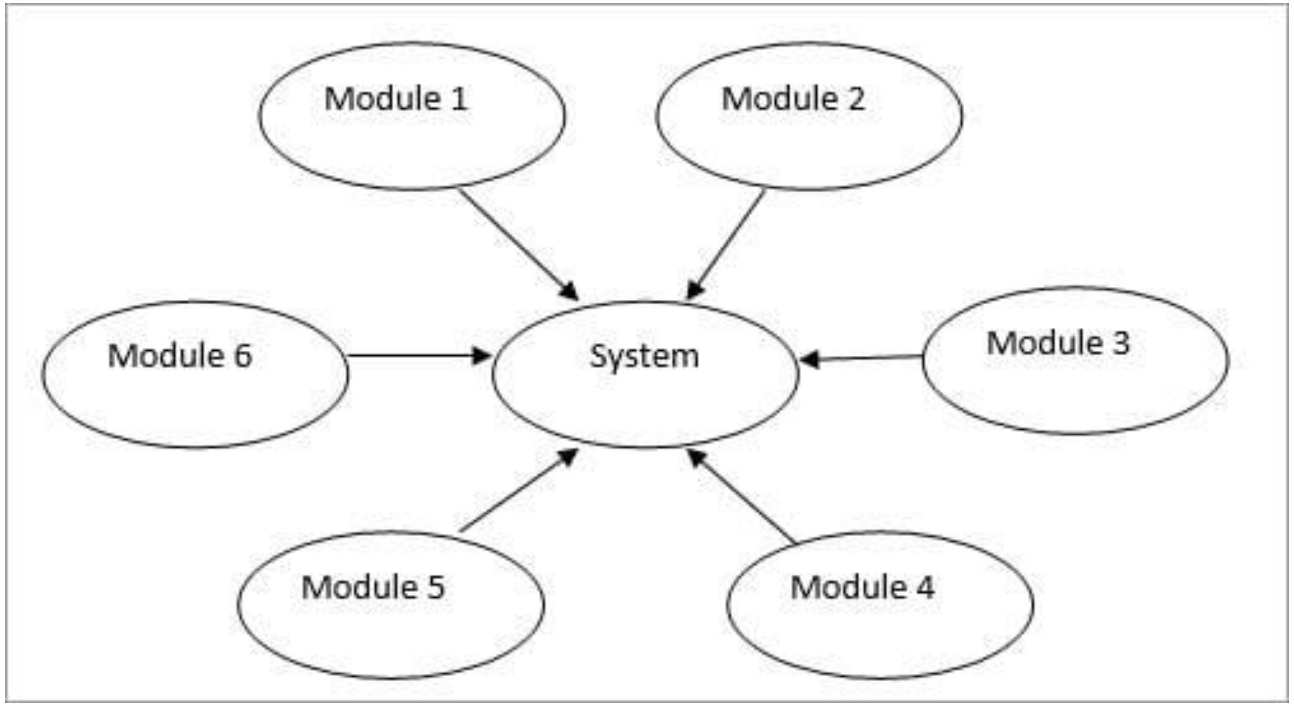
- Individual units are combined and tested as a group
- Can be performed by both the developers and the testers
- Focuses mainly on the interfaces & flow of data/information between the modules

Approaches/Methodologies/Strategies of Integration Testing:

- Big Bang Approach
- Incremental Approach
  - Top Down Approach
  - Bottom Up Approach
  - Sandwich Approach

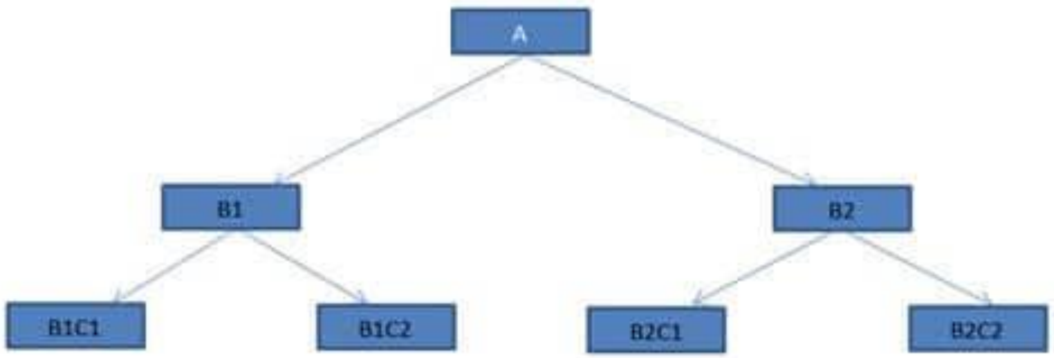
Integration Testing: Big Bang Approach

- Integrates all the modules.
- Modules are not integrated one by one.
- Difficult to find out which module has caused the issue.
- Good for small systems.



Integration Testing: Bottom-up Approach

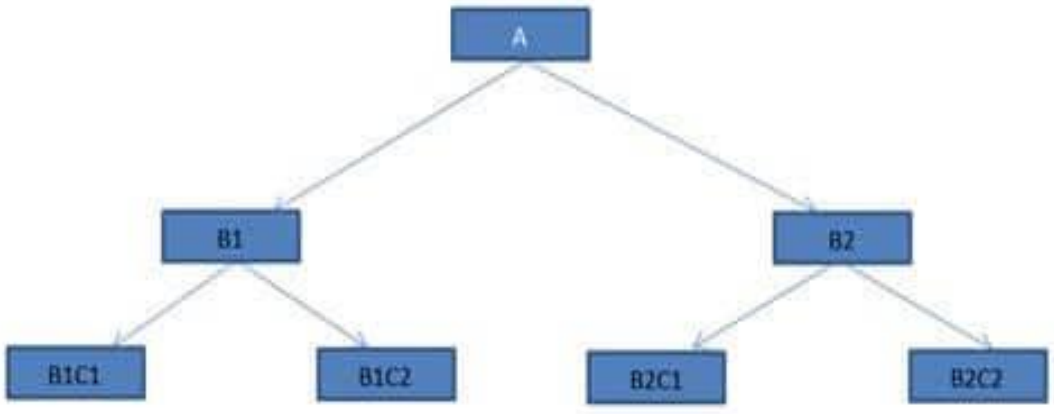
- starts from the lowest or the innermost unit of the application, and gradually moves up.



- B1C1, B1C2 & B2C1, B2C2 are the lowest module which is unit tested
- B1 & B2 are not yet developed.
- The functionality of Module B1 and B2 is that it calls the modules B1C1, B1C2 & B2C1, B2C2.
- Since B1 and B2 are not yet developed, some programs or a "stimulator" is needed which will call the B1C1, B1C2 & B2C1, B2C2 modules. These stimulator programs are called **DRIVERS**.

Integration Testing: Top-down Approach

- Starts from the topmost module and gradually progress towards the lower modules.



- Testing starts from Module A
- Lower modules B1 and B2 are not yet ready
- So in order to test the topmost modules A, we develop "**STUBS**".
- "**Stubs**" can be referred to as code a snippet which accepts the inputs/requests from the top module and returns the results/ response.

System Testing

- Falls under the **black box testing.**

- Independent Testers perform System Testing.

- Testing of a complete and fully integrated software product.

System Testing: Some types

- ✓ **Usability Testing:** Mainly focuses on the user's ease to use the application
- ✓ **Load Testing:** To know if a software will perform under real-life loads.
- ✓ **Regression Testing:** To make sure none of the changes made over the course of the development process have caused new bugs.
- ✓ **Recovery Testing:** To verify that software successfully recover from possible crashes.
- ✓ **Migration Testing:** To check if software can be moved from older system to current system without any issues.
- ✓ **Functional Testing:** Testers might make a list of additional functionalities that a product could have to improve it during functional testing.
- ✓ **Hardware/Software Testing:** Tester focuses his/her attention on the interactions between the hardware and software during system testing.

Acceptance Testing

- A system is tested for acceptability.
- To evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery.
- Black box testing method
- Performed by the client
- To make sure that developers had implemented the software according to what clients told them to do.

Flow Graph Notation

- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Cyclomatic Complexity

- Software metric used to indicate the complexity of a program
- It is computed using the Control Flow Graph of the program

Cyclomatic Complexity	Meaning
1 – 10	• Structured and Well Written Code • High Testability • Less Cost and Effort
10 – 20	• Complex Code • Medium Testability • Medium Cost and Effort
20 – 40	• Very Complex Code • Low Testability • High Cost and Effort
>40	• Highly Complex Code • Not at all Testable • Very High Cost and Effort

Calculating Cyclomatic Complexity

- **Method-01:**

  Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1
- **Method-02:**

  Cyclomatic Complexity = E – N + 2*P

Here,

- E = Total number of edges in the control flow graph
- N = Total number of nodes in the control flow graph
- P = Connected Component

- **Method-03:**

  Cyclomatic Complexity = P + 1

Here,

- P = Total number of predicate nodes contained in the control flow graph

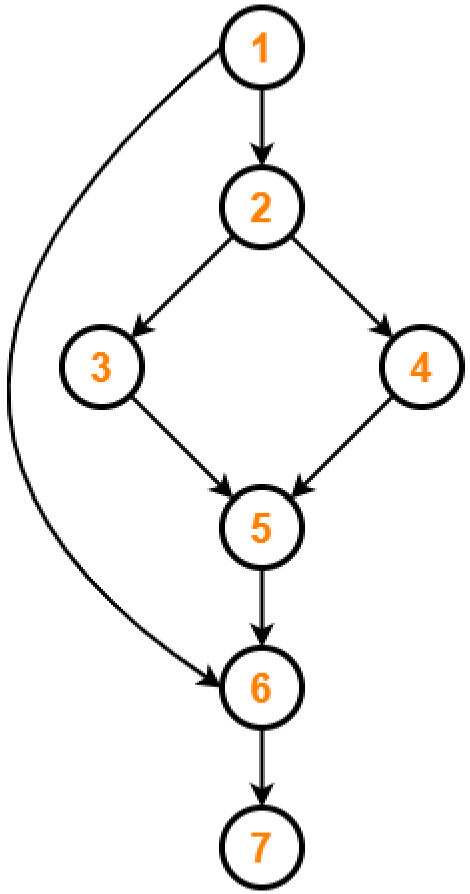
- **Note-**
- Predicate nodes are the conditional nodes.
- They give rise to two branches in the control flow graph.

Practice Problem on Cyclomatic Complexity

Calculate cyclomatic complexity for the given code-

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
END IF
END IF
PRINT A
```

Control Flow Graph



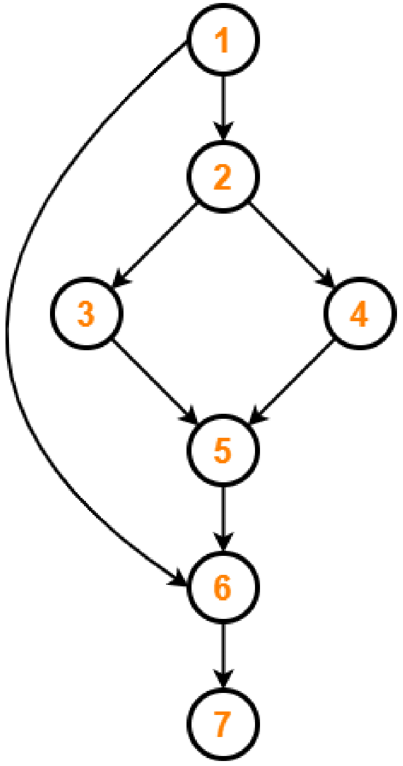
Control Flow Graph

Practice Problem on Cyclomatic Complexity

Calculate cyclomatic complexity for the given code-

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
END IF
END IF
PRINT A
```

Control Flow Graph



Control Flow Graph

Method 01:

Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1

= 2 + 1

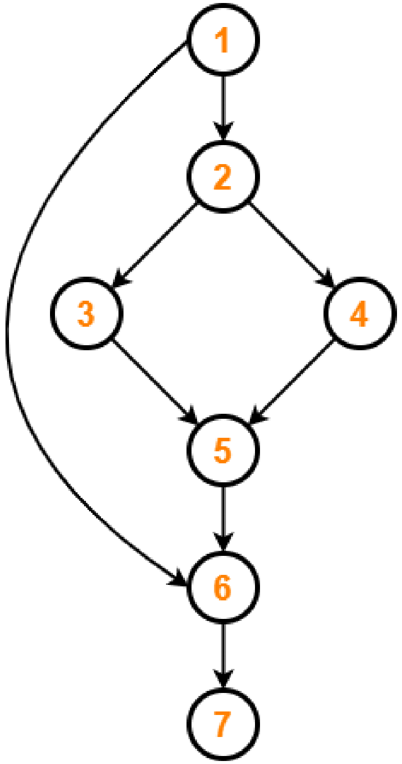
= 3

Practice Problem on Cyclomatic Complexity

Calculate cyclomatic complexity for the given code-

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
END IF
END IF
PRINT A
```

Control Flow Graph



Control Flow Graph

Method 02:

Cyclomatic Complexity = E – N + 2

= 8 – 7 + 2

= 3

E = No. of edges in the CFG

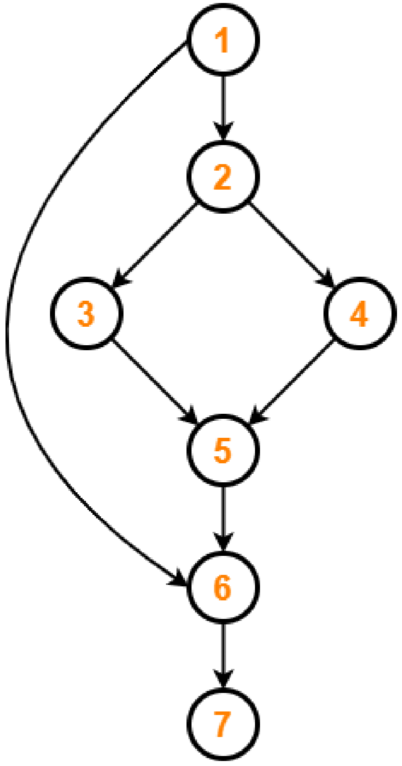
N = No. of nodes in the CFG

Practice Problem on Cyclomatic Complexity

Calculate cyclomatic complexity for the given code-

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
END IF
END IF
PRINT A
```

Control Flow Graph



Control Flow Graph

Method 03:

Cyclomatic Complexity = P + 1
= 2 + 1
= 3

P = predicate nodes

Which are having two outgoing edges

Node 1 and node 2 have 2 outgoing edges

So, P = 2

Practice Problem on Cyclomatic Complexity

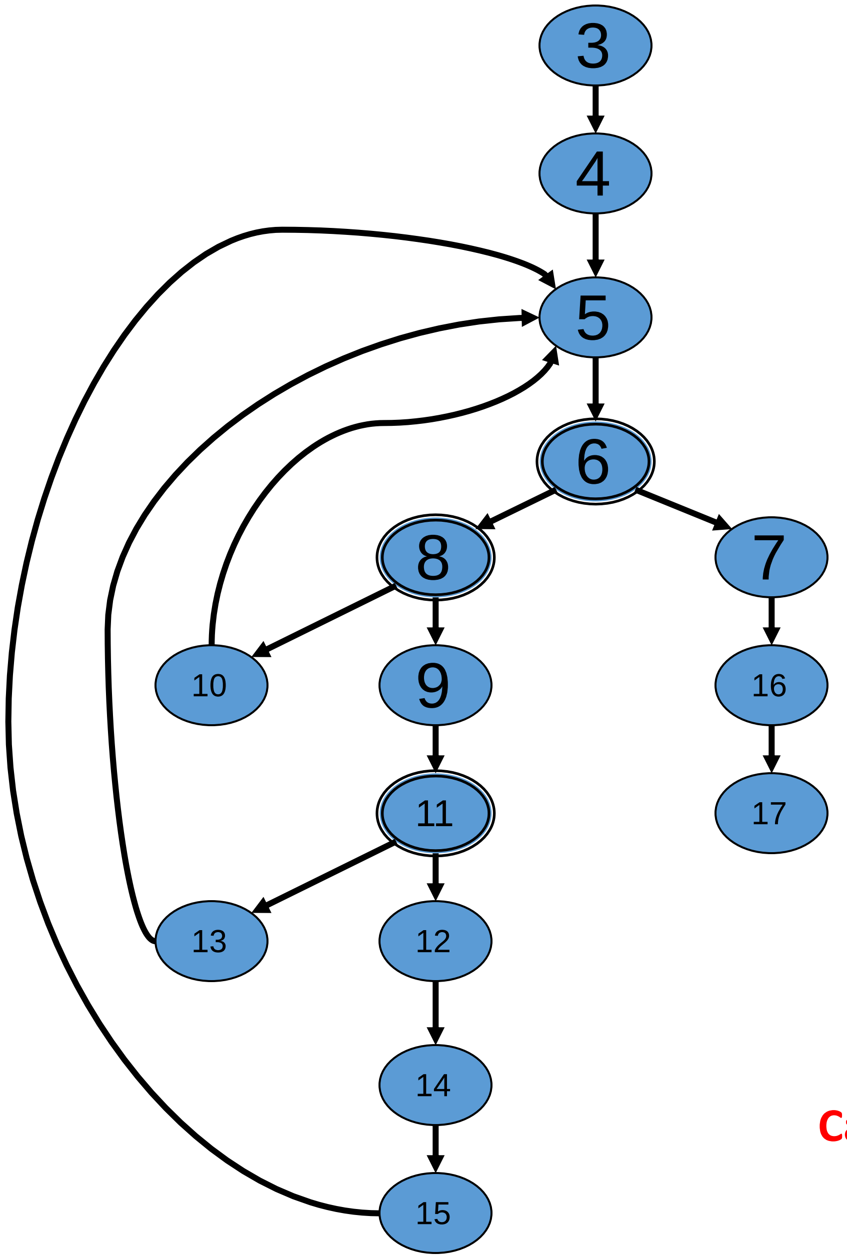
```
1   int functionY(void)
2   {
3      int x = 0;
4      int y = 19;

5   A: x++;
6      if (x > 999)
7         goto D;
8      if (x % 11 == 0)
9         goto B;
10     else goto A;

11  B: if (x % y == 0)
12        goto C;
13     else goto A;

14  C: printf("%d\n", x);
15     goto A;

16  D: printf("End of
    list\n");
17     return 0;
18  }
```

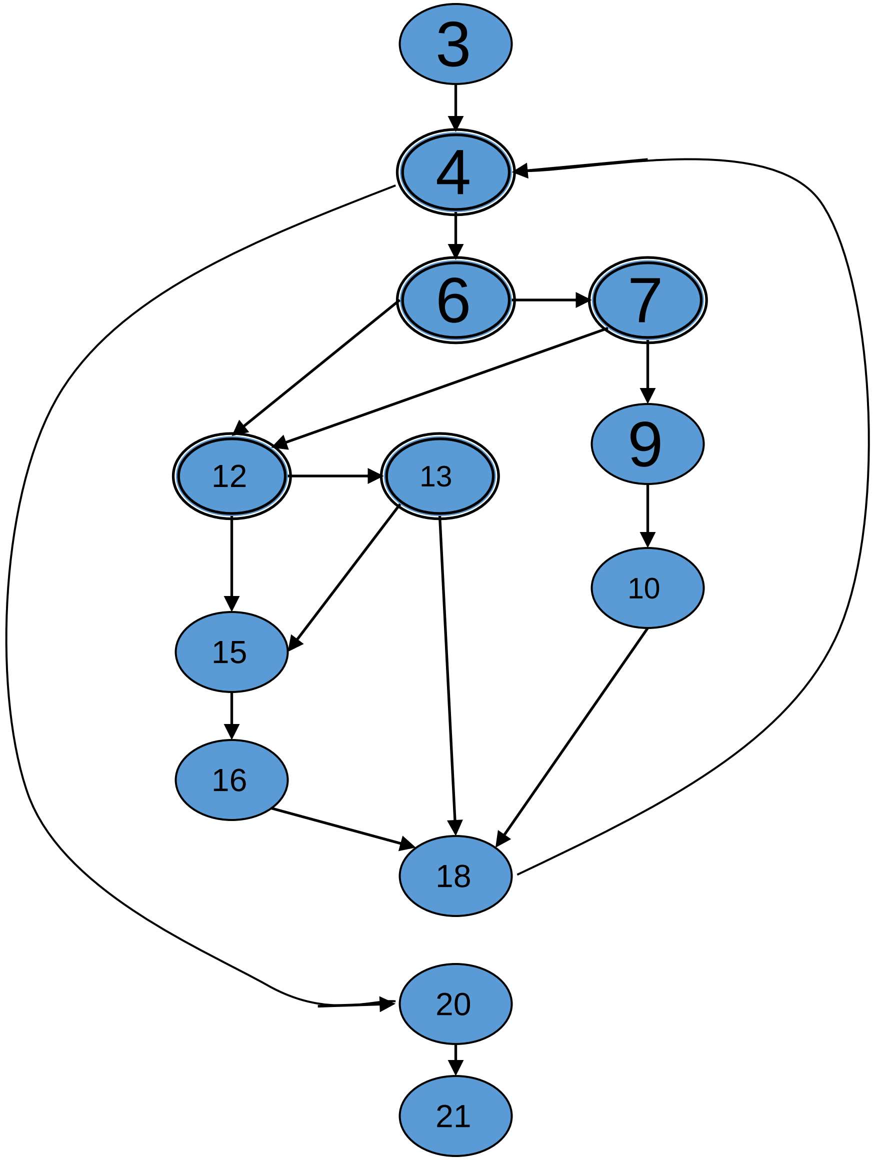


Calculate Cyclomatic Complexity

Practice Problem on Cyclomatic Complexity

```
int functionZ(int y)
{
int x = 0;
while (x <= (y * y))
    {
    if ((x % 11 == 0) &&
         (x % y == 0))
        {
        printf("%d", x);
        x++;
        } // End if
    else if ((x % 7 == 0) ||
                (x % y == 1))
        {
        printf("%d", y);
        x = x + 2;
        } // End else
    printf("\n");
    } // End while

printf("End of list\n");
return 0;
} // End functionZ
```



Calculate Cyclomatic Complexity

Practice Problem on Cyclomatic Complexity

Cyclomatic Complexity Flow Graph Example | Gate Vidyalay

Smoke Testing

- Testing the most important and critical parts of a software
- Testing is performed after each build
- Do not check any function in depth
- If software passes smoke test, then we can send it to further testing otherwise there will be waste of money and time
- Generally performed by developer or QA team