Coding and Testing

Courtesy:

Roger Pressman, Ian Sommerville &

Prof Rajib Mall

Coding:

- The objective of the coding phase is to transform the design of a system into code in a high-level language , and then to unit test this code.
- Software development organization adhere to well defined and standard style of coding- **Coding Standards.**
  - A coding standard gives a uniform appearance to the codes written by different engineers.
  - It facilitates code understanding and code reuse.
  - It promotes good programming practices.

Coding:

- **Coding standards** are mandatory for the programmers to follow.
- Compliance of their code to coding standards is verified during **code inspection.**
- Any code that does not conform to the coding standards is rejected during **code review** and the code is reworked by the concerned programmer.
- In contrast, **coding guidelines** provides some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

Coding Standards and Guidelines:

- Organization usually develop their own coding standards and guidelines depending on
  - what suits their organization best and
  - based on the specific types of software they develop.
- **Representative Coding Standards:**
  - Rules for limiting the use of global.
  - Standard headers for different modules
  - Naming conventions for global variables, local variables, and constant identifiers.
  - Convention regarding error return values and exception handling mechanisms

Example: Standard headers for different modules

- An example of header format that is being used in some organization:
  - Name of the module
  - Date on which the module was created
  - Author's name
  - Modification history
  - Synopsis of the module
  - Different functions supported in the module, along with their input/output parameters
  - Global variables accessed/modified by the module

Coding Guidelines:

- The representative coding guidelines followed by the Software development organization are as ..
  - Do not use coding style that is too difficult to understand.
  - Do not use identifier for multiple purposes
  - Code should be well documented
  - Length of any function should not exceed 10 source lines
  - Do not use GO TO statements

Code Review:

- Eliminating an error from code involves: **Testing (detecting failures), debugging (Locating errors), and then correcting the errors.**
- Code review and testing are both effective defect removal mechanisms.
  - The **code review** is much more cost effective strategy to eliminate errors from code as it directly detect errors.
  - **Testing** only helps detect failures and significant effort is needed to locate the error during debugging.
- The following two types of reviews are carried out on the code of a module:
  - Code inspection
  - Code walkthrough

Code Walkthrough:

- Code walkthrough is an **informal code analysis technique.**
- The main objective of code walkthrough is **to discover the algorithmic and logical errors** in the code.
- Discussion in the walkthrough meeting should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
- The size of the team performing code walkthrough should consist of three to seven members and **managers should not be a part of walkthrough meetings.**

Code Inspection:

- During code inspection, the code is examined to check for the presence of some common programming errors.
- The principal aim of code inspection is
  - to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and
  - to check whether coding standards have been adhered to.
- The list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

Code Inspection:

- The following is the checklist of some classical programming errors which can be used during code inspection:
  - Use of uninitialized variables
  - Use of incorrect logical operators or incorrect precedence among operators.
  - Incomplete assignments
  - Jumps into loops
  - Non-terminating loops
  - Array indices out of bounds
  - Improper storage allocation and deallocation
  - Mismatch between actual and formal parameters in procedure calls
  - Dangling reference caused when reference memory has not been allocated

Characteristics of Testable Software

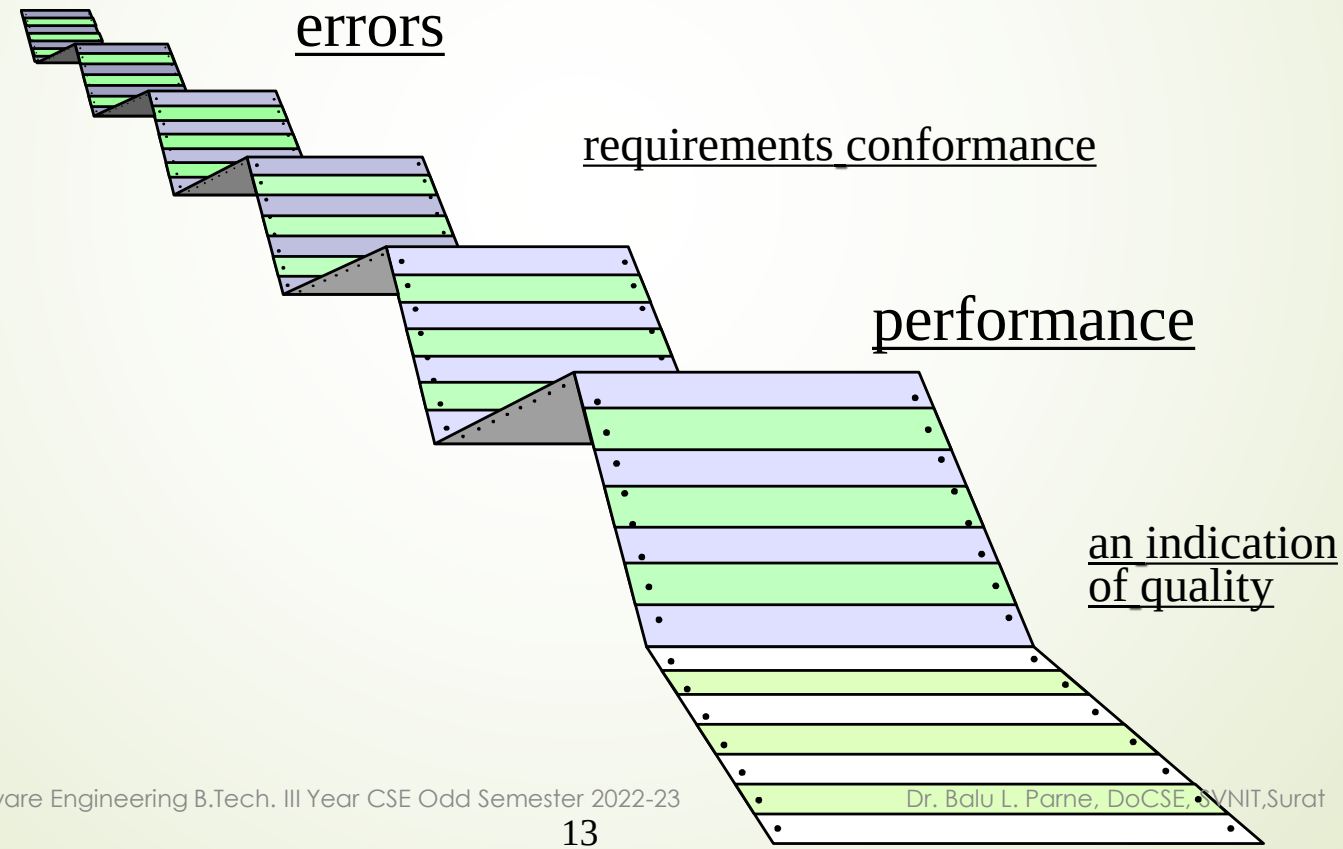
- **Operability**
  - The better it works (i.e., better quality), the easier it is to test
- **Observability**
  - Incorrect output is easily identified; internal errors are automatically detected
- **Controllability**
  - The states and variables of the software can be controlled directly by the tester
- **Decomposability**
  - The software is built from independent modules that can be tested independently

Characteristics of Testable Software

- **Simplicity**
  - The program should exhibit functional, structural, and code simplicity
- **Stability**
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- **Understandability**
  - The architectural design is well understood; documentation is available and organized

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.



Who Tests the Software?

developer

independent tester

Understands the system
but, will test "gently"
and, is driven by ***"delivery".***

Must learn about the system,
but, will attempt to break it
and, is driven by "**quality".**

Software Testing:

- Software Testing, when done correctly, **can increase overall software quality by testing that the product conforms to its requirements.**
- After generating source code, **the software must be tested to uncover and correct as many errors as possible before delivering to customer.**
- Testing is intended **to show that a program does what it is intended to do** and **to discover program defects before it is put into use.**
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.

Testing Objectives:

PREVENT DEFECTS

Efficient testing helps preventing defects and that helps in providing an error-free application.

EVALUATE WORK PRODUCTS

To evaluate work products such as requirements, user stories, design, and code

FIND FAILURE AND DEFECTS

To find failures and defects. Defects should be identified as early in the test cycle as possible..

VERIFY REQUIREMENT

To verify whether all specified requirements have been fulfilled

SHARE INFORMATION TO STAKEHOLDERS

To provide enough information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object

VALIDATE TEST OBJECT

To validate whether the test object is complete and works as the users and other stakeholders expect

REDUCE RISK

To reduce the level of risk of inadequate software quality (e.g., previously undetected failures occurring in operation)

BUILD CONFIDENCE

To build confidence in the level of quality of the test object

- **To evaluate the work products such as requirements, design, user stories, and code:**
  - The work products such as **Requirement document, Design, and User Stories** should be verified before the developer picks it up for development.
  - Identifying any ambiguity or contradicting requirements at this stage saves considerable development and test time.
  - The **static analysis of the code** (reviews, walk-thru, inspection, etc.) happens before the code integrates/is ready for testing.
- **To verify the fulfillment of all specified requirements:**
  - This objective reveals the fact that the essential elements of testing should be to fulfill the customer's needs.
  - Developing all the test cases, regardless of the testing technique **ensures verification of the functionality for every executed test case.**

- **To validate if the test object is complete and works as per the expectation of the users and the stakeholders:**
  - Testing **ensures the implementation of requirements along with the assurance that they work as per the expectation of users.**
  - It usually employs various types of testing techniques, i.e., black box, white box, etc.
  - Every business considers the customer as the king. Thus, **the customer's satisfaction is must for any business.**
- **To build confidence in the quality level of the test object:**
  - One of the critical objectives of software testing is **to improve software quality. High-Quality software means a lesser number of defects.**
  - In other words, the more efficient the testing process is, the fewer errors you will get in the end product. Which, in turn, will increase the overall quality of the test object.

- **To prevent defects in the software product:**
  - One of the objectives of software testing is **to avoid the mistakes in the early stage of the development.** Early detection of errors significantly reduces the cost and effort.
  - Efficient testing helps in providing an error-free application. If you prevent defects, it will result in reducing the overall defect count in the product, which further ensures a high-quality product to the customer.
- **To find defects in the software product:**
  - Another essential objective of software testing is **to identify all defects in a product.**
  - The main motto of testing is to find maximum defects in a software product while validating whether the program is working as per the user requirements or not.

- **To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object**
  - The purpose of testing is **to provide complete information to the stakeholders about technical or other restrictions, risk factors, ambiguous requirements, etc.**
  - It can be in the form of test coverage, **testing reports covering details like what is missing, what went wrong.**
  - The aim is to be transparent and make stakeholders fully understand the issues affecting quality.

- **To comply with contractual, legal, or regulatory requirements or standards, and to verify the test object's compliance with such requirements or standards:**
  - This objective ensures that software developed for a specific region **must follow the legal rules and regulations of that region.**
  - Moreover, the software product must be compatible with the national and international standards of testing.
  - **ISO/IEC/IEEE 29119** standards deal with the software testing concept.

- **To reduce the level of risk of insufficient software quality:**
  - The possibility of loss is also known as risk. The objective of software testing is to reduce the occurrence of the risk.
  - Each software project is unique and contains a **significant number of uncertainties** from different perspectives, such as **market launch time, budget, the technology chosen, implementation, or product maintenance.**
  - If we do not control these uncertainties, it will impose potential risks not only during the development phases but also during the whole life cycle of the product. So, the primary objective of software testing is to integrate the Risk management process to identify any risk as soon as possible in the development process.

Testing Guidelines:

- Development team should avoid testing the software
- Start as early as possible
- Prioritize sections
- The time available is limited
- Testing must be done with unexpected and negative inputs
- Inspecting test results properly
- Validating assumptions
- Software can never be 100% bug-free

Verification and Validation:

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance.
- **Verification (Are we building the product right?)**
  - The set of activities that ensure that software correctly implements a specific function or algorithm.
- **Validation (Are we building the right product?)**
  - The set of activities that ensure that the software that has been built is traceable to customer requirements.

Verification and Validation:

- **Verification** is the process of determining:
  - Whether output of one phase of development conforms to its previous phase.
  - Verification is concerned with phase containment of errors.

- **Validation** is the process of determining:
  - Whether a fully developed system conforms to its SRS document.
  - The aim of validation is that the final product is error free.

Verification and Validation:

- Ex: Lets say we are writing a program for addition.
  - **a+b = c**
- **Verification** (Are we building the product right**?**)
  - Are we getting **some** output for **a+b?**
- **Validation** (Are we building the right product**?**)
  - Are we getting **correct** output for **a+b?**

Verification and Validation:

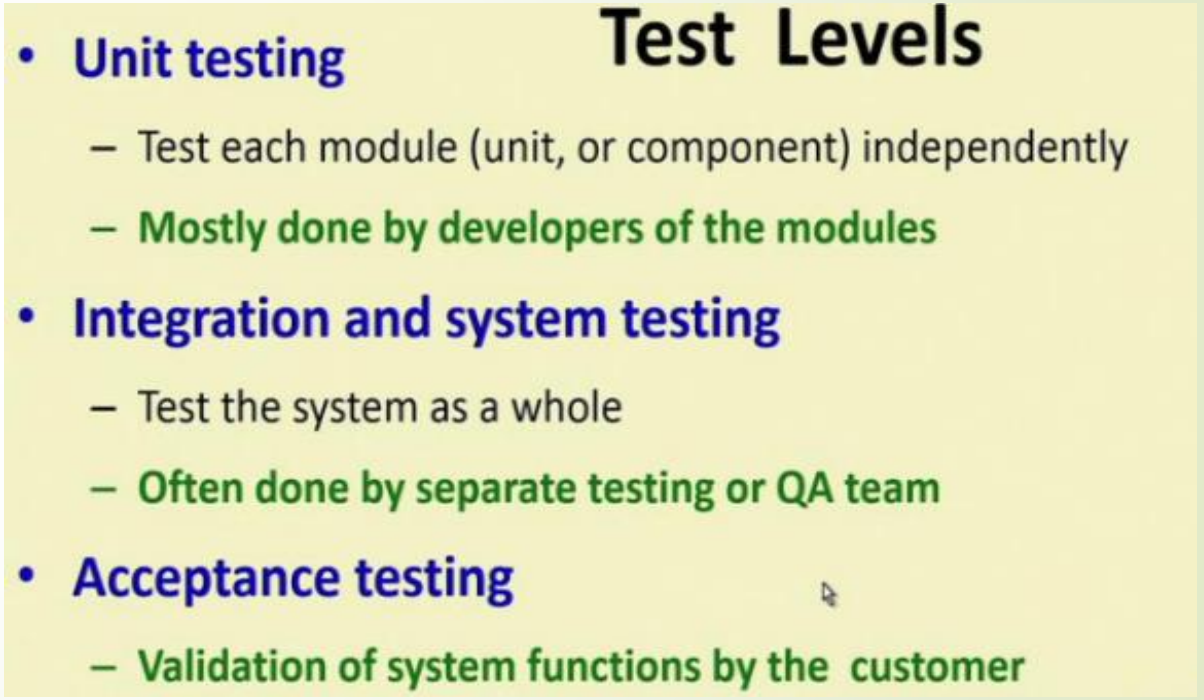
Verification and Validation Techniques

- Review
- Simulation
- Unit testing
- Integration testing
- System testing

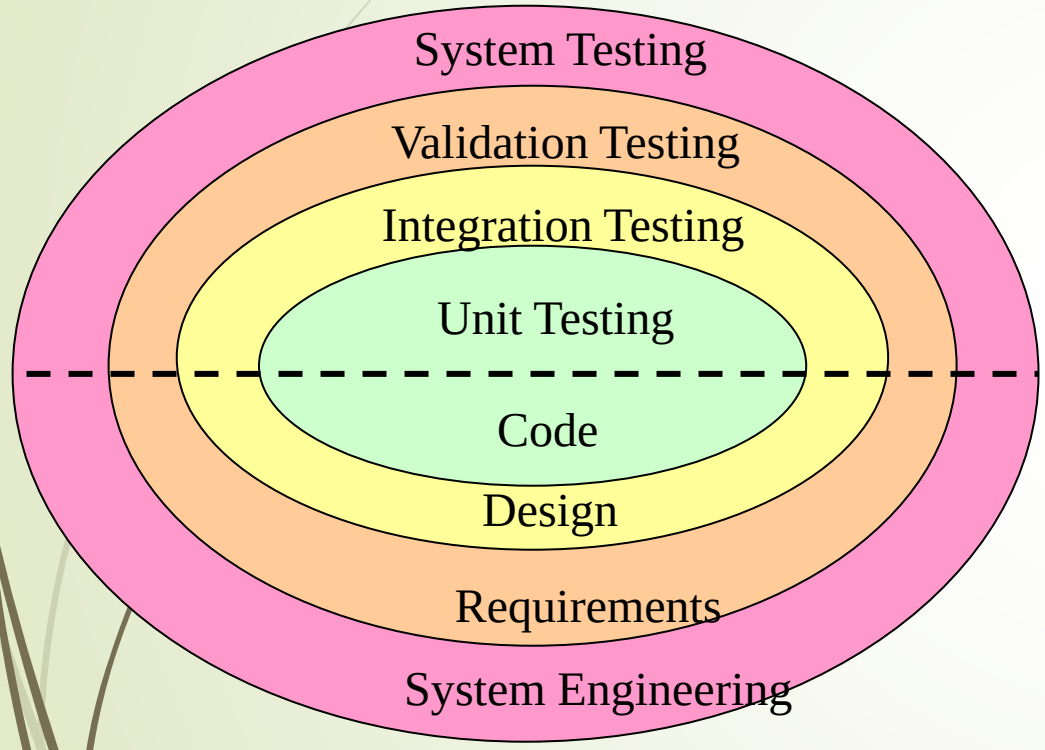
Verification	Validation
Are you building it right?	Have you built the right thing?
Checks whether an artifact conforms to its previous artifact.	Checks the final product against the specification.
Done by developers.	Done by Testers.
Static and dynamic activities: reviews, unit testing.	Dynamic activities: Execute software and check against requirements.

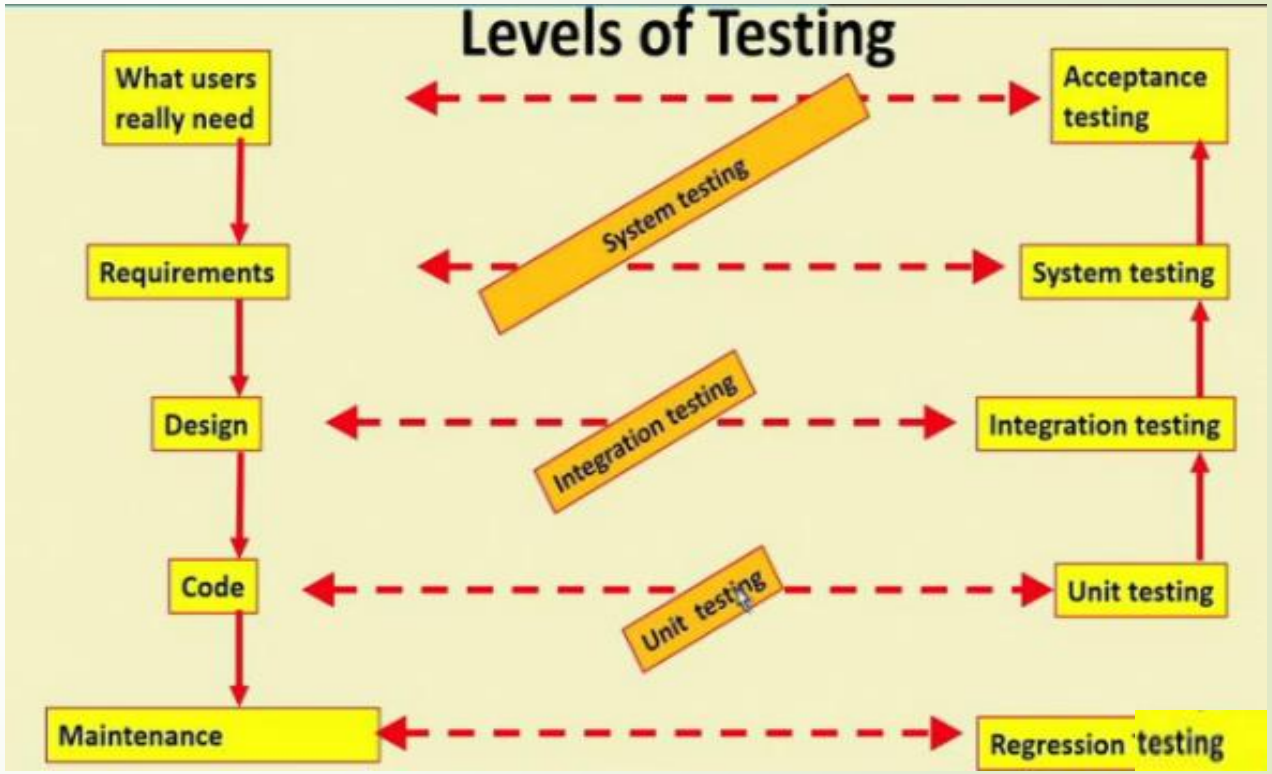
4 Testing Levels:

- Software tested at 4 levels:
  - Unit Testing
  - Integration Testing
  - System Testing
  - Regression Testing

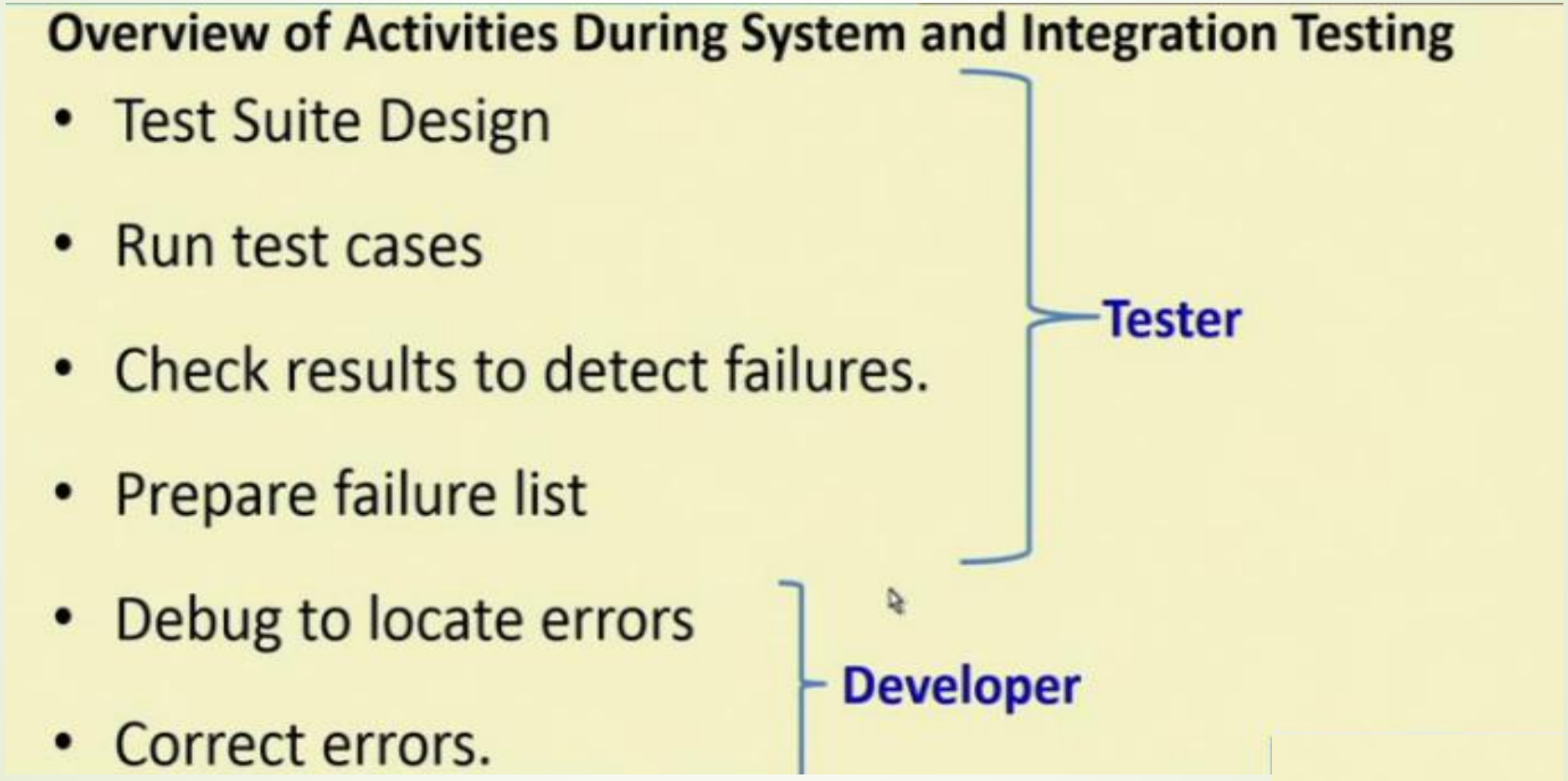


Levels of Testing:





Overview of Activities During System and Integration Testing



Test Cases:

- Each test case typically tries to establish correct working of some functionality:
  - Execute (covers) some program elements.
  - For certain restricted types of fault, fault based testing can be used.
- **Test Data:**
  - Inputs used to test the system.
- **Test Cases:**
  - Inputs to test the system.
  - State of the software, and
  - The predicted outputs from the inputs

Test Cases:

- A test case is the triplet (I, S, O)
  - I is the data to be input to the system.
  - S is the state of the system at which the data will be input.
  - O is the expected output of the system.

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the **test suite.**

Negative Test Cases:

- Helps to ensure that the application gracefully handles invalid and unexpected user inputs and the application does not crash.
- **Example:**
  - If user types letter in a numeric field, it should not crash but politely display the message: **"Incorrect data type, please enter a number"**

Test Cases:

- A test case contains:
  - A sequence of Steps describing actions to be performed,
  - Test data to be used
  - An expected response for each action performed.
- Test Cases are **written based on Business and Functional/Technical requirements, use cases and Technical design documents.**
- There can be **1:1 or 1:N or N:1 or N:N relationship** between requirements and Test cases
- The level of details specified in test cases will vary depending on Organizations, Projects and also on the Test Case Template used OR on the Test Management tool being used in the project.

Test Cases:

- Construction of Test Cases also helps in:
  - Finding issues or gaps in the requirements
  - Technical design itself.
- As Test Case construction activity would make tester to think through different possible Positive and Negative scenarios.
  - It is test cases against which tester will verify the application is working as expected.
- Number of test cases to be created depends on the size, complexity and type of testing being performed.

Test Cases Example:

- Test case for ATM
  - TC 1 :- successful card insertion.
  - TC 2 :- unsuccessful operation due to wrong angle card insertion.
  - TC 3:- unsuccessful operation due to invalid account card.
  - TC 4:- successful entry of pin number.
  - TC 5:- unsuccessful operation due to wrong pin number entered 3 times.
  - TC 6:- successful selection of language.
  - TC 7:- successful selection of account type.
  - TC 8:- unsuccessful operation due to wrong account type selected w.r.t. inserted card.
  - TC 9:- successful selection of withdrawal option.
  - TC 10:- successful selection of amount.
  - TC 11:- unsuccessful operation due to wrong denominations.
  - TC 12:- successful withdrawal operation.
  - TC 13:- unsuccessful withdrawal operation due to amount greater than balance.
  - TC 14:- unsuccessful due to lack of amount in ATM.

Why Design Test Cases?

- Exhaustive testing of any non-trivial system is impractical:
  - Input data domain is extremely large.
- Design an **optimal test suite** meaning:
  - Of reasonable size, and
  - Uncovers as many errors as possible
- **If test cases are selected randomly:**
  - Many test cases would not contribute to the significance of the test suite,
  - Would only detect errors that are already detected by other test cases in the suite.
- **Therefore, the number of test cases in a randomly selected test suite:**
  - **Does not indicate the effectiveness of testing**

Design of Test Cases:

- Testing a system using a large number of randomly selected test cases:
  - Does not mean that most errors in the system will be uncovered.
- Consider the function that find maximum of two integers x and y.

```
If (x>y) max = x;
else max = x;
```

- Systematic approaches are required to design an effective test suite:
  - Each test case in the suite should target different faults

Black-box Testing:

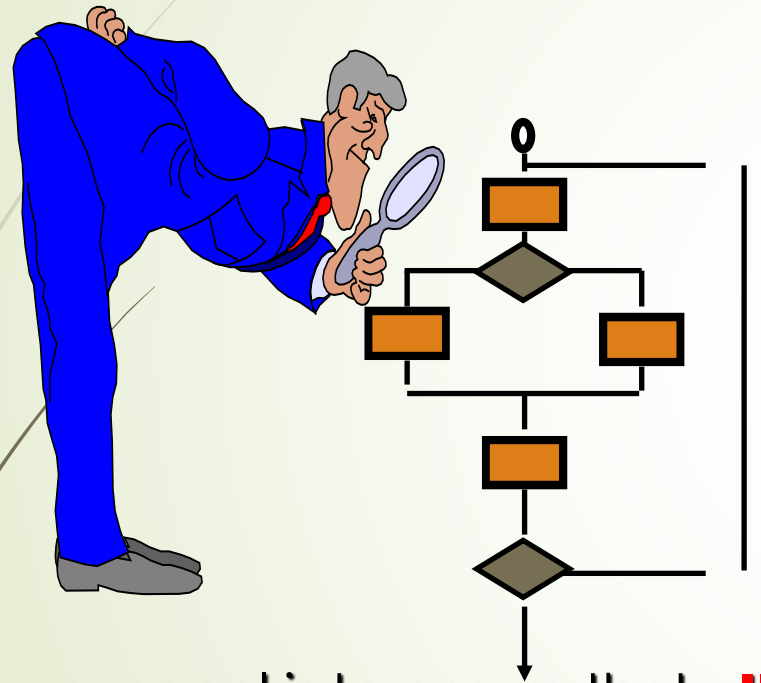
- Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free.
- **Includes tests that are conducted at the software interface.**
- **Not concerned with internal logical structure of the software .**

White-box Testing:

- Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised.
- **Involves tests that concentrate on close examination of procedural detail.**
- **Logical paths through the software are tested.**
- Test cases exercise specific sets of conditions and loops.

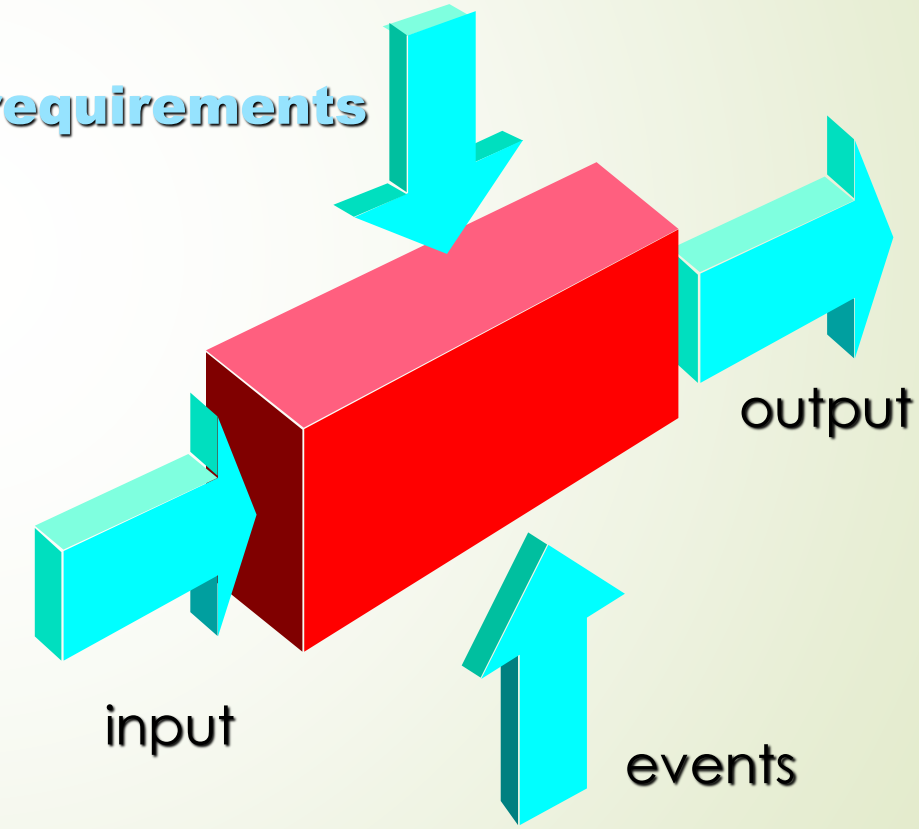
Software Testing

White-Box Testing



... our goal is to ensure that **all** statements and conditions have been executed at least once ...

Black-Box Testing



White-box Testing

White-box Testing:

- **Uses the control structure part** of component-level design **to derive the test cases**
- These test cases
  - Guarantee that **all independent paths within a module have been exercised at least once**
  - **Exercise all logical decisions** on their true and false sides.
  - **Execute all loops at their boundaries** and within their operational bounds.
  - **Exercise internal data structures** to ensure their validity.

Basis Path Testing:

- Basis path testing is a white-box testing technique proposed by **Tom McCabe.**
- Enables the test case designer to derive a logical complexity measure of a procedural design.
- Uses this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

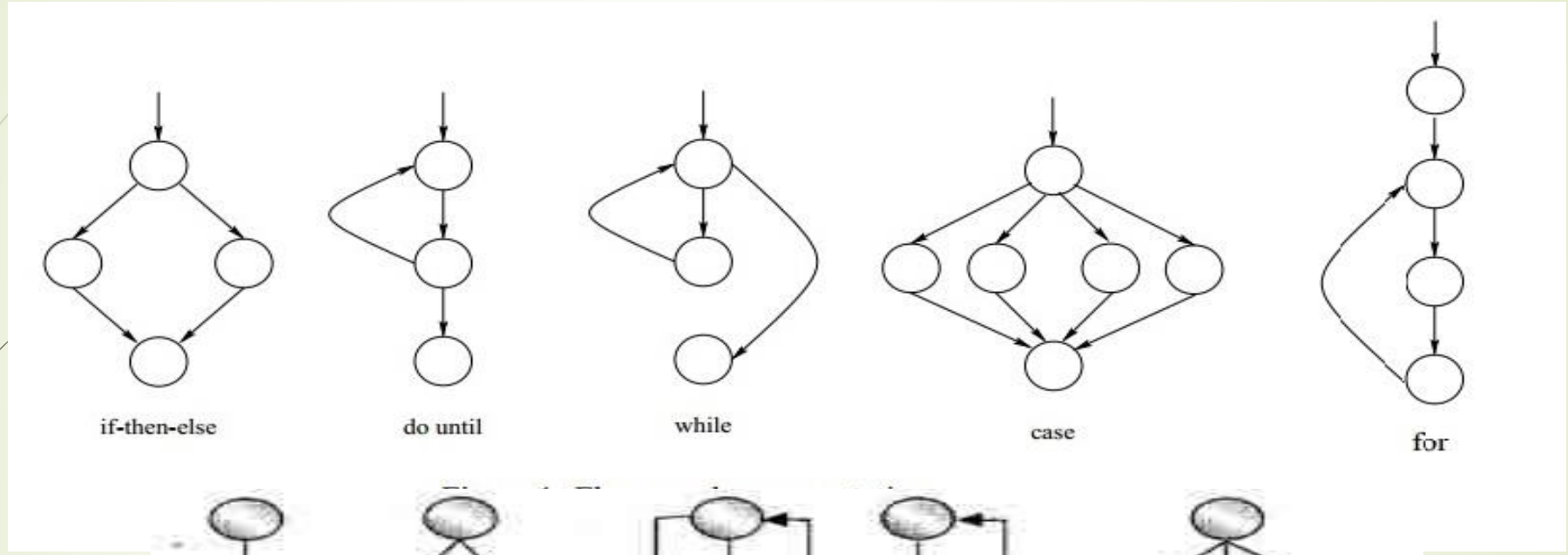
Flow Graph Notation:

- A flow graph should be drawn only when the logical structure of a component is complex.
- The flow graph allows to trace program paths more readily.
- A simple notation for the representation of control flow, called a flow graph (or program graph) are introduced, which will depicts logical control flow.
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements.

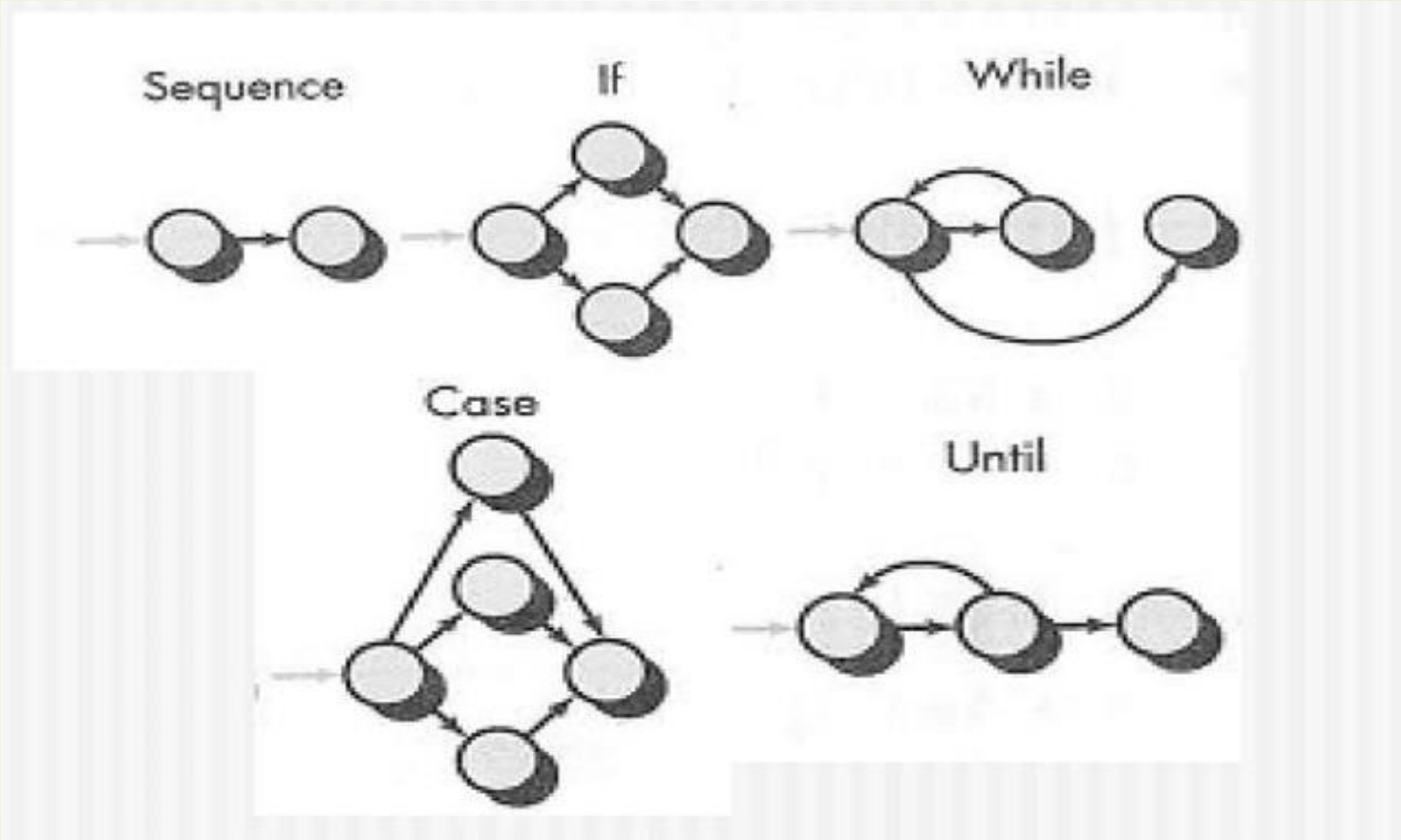
Flow Graph Notation:

- The arrows on the flow graph, called edges or links , represents flow of control and are analogous to flowchart arrow.
  - 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.

Flow Graph Notation:



Flow Graph Notation:

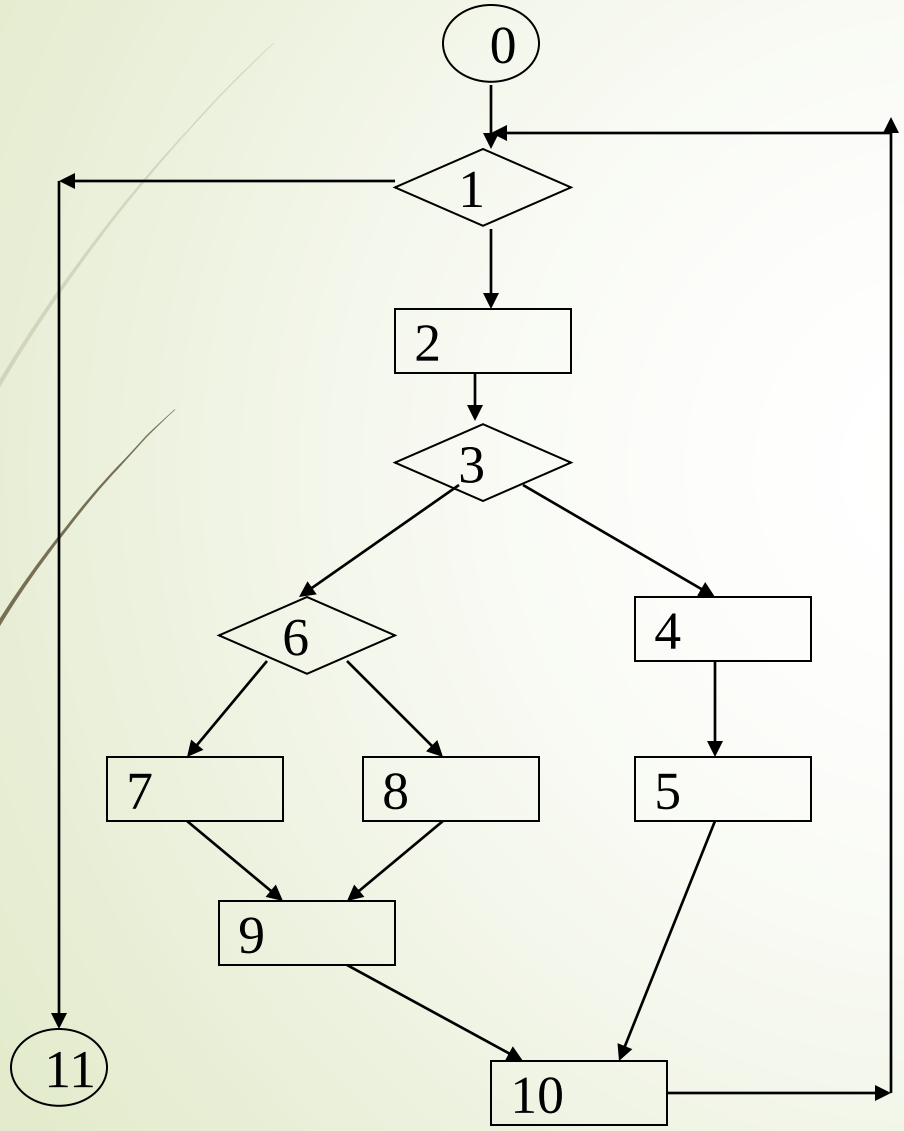


Flow Graph Notation:

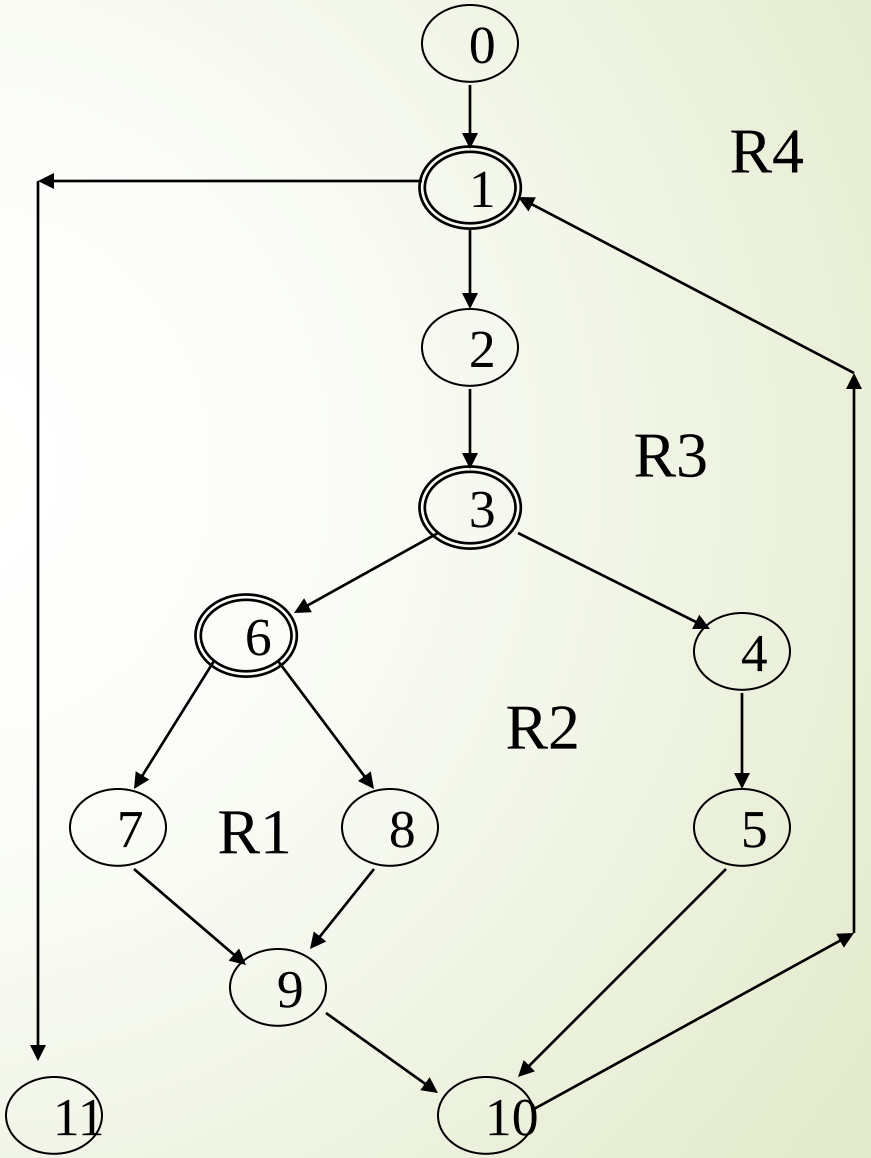
- A node containing a simple conditional expression is referred to as a predicate node.
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- When compound conditions are encountered in a procedural design, the generation of flow graphs become slightly more complicated.
- Compound condition occurs when one or more Boolean operators is present in conditional statement.

Flow Graph Example

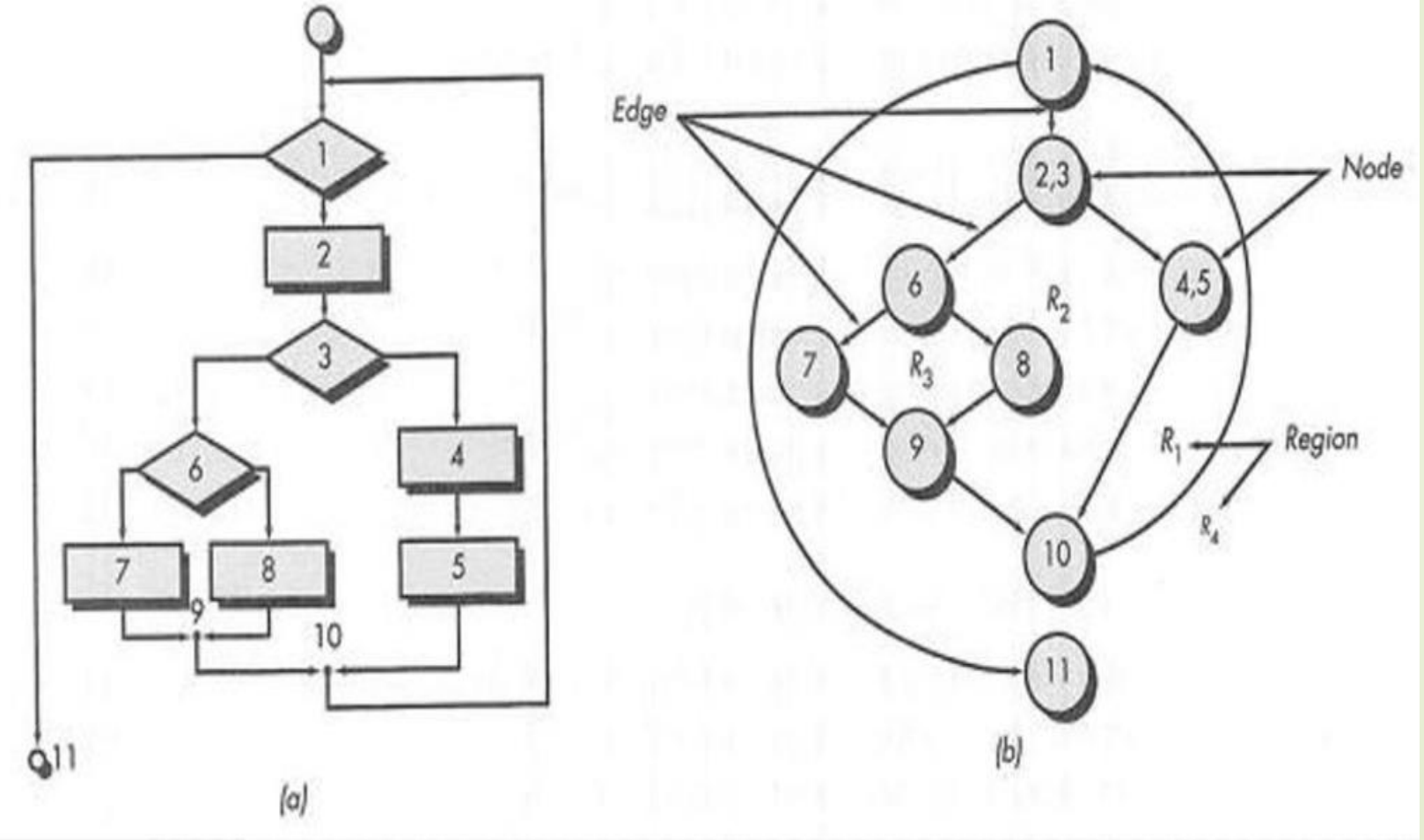
FLOW CHART



FLOW GRAPH



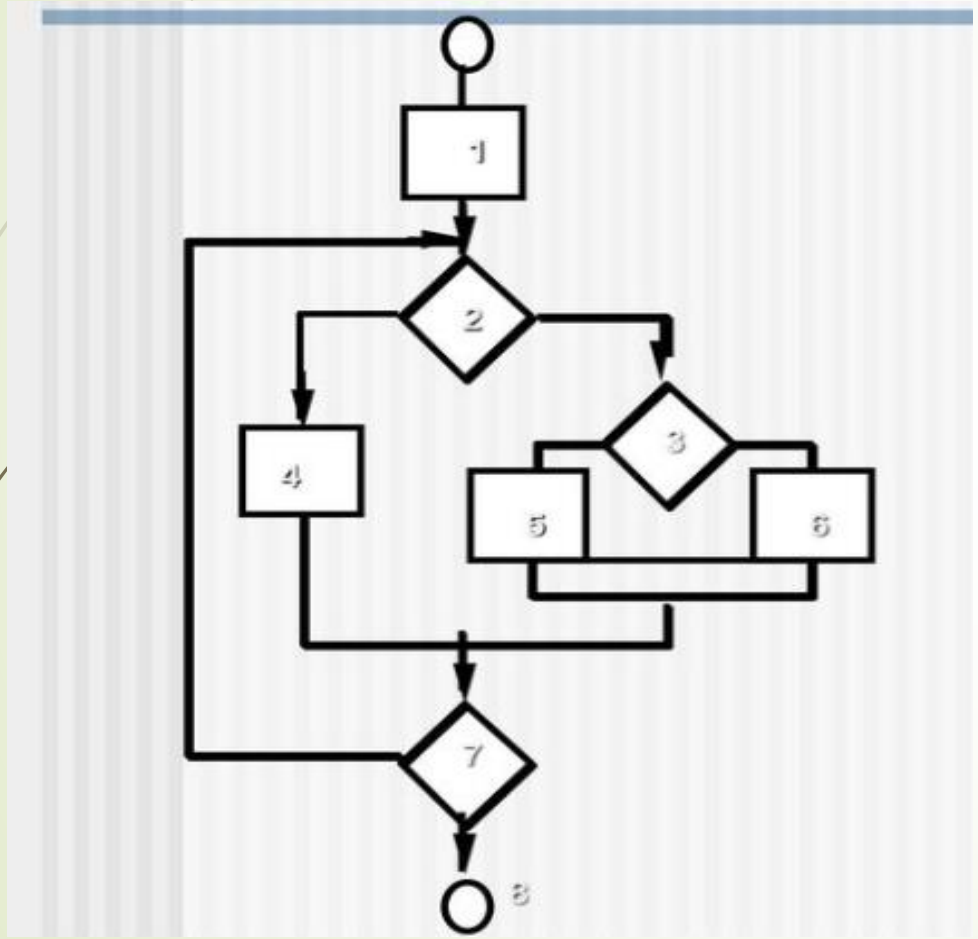
Flow chart and Flow Graph:



Independent Program Paths:

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes).
- Must move along at least one edge that has not been traversed before by a previous path.
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity.

Independent Program Paths:



Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Cyclomatic Complexity:

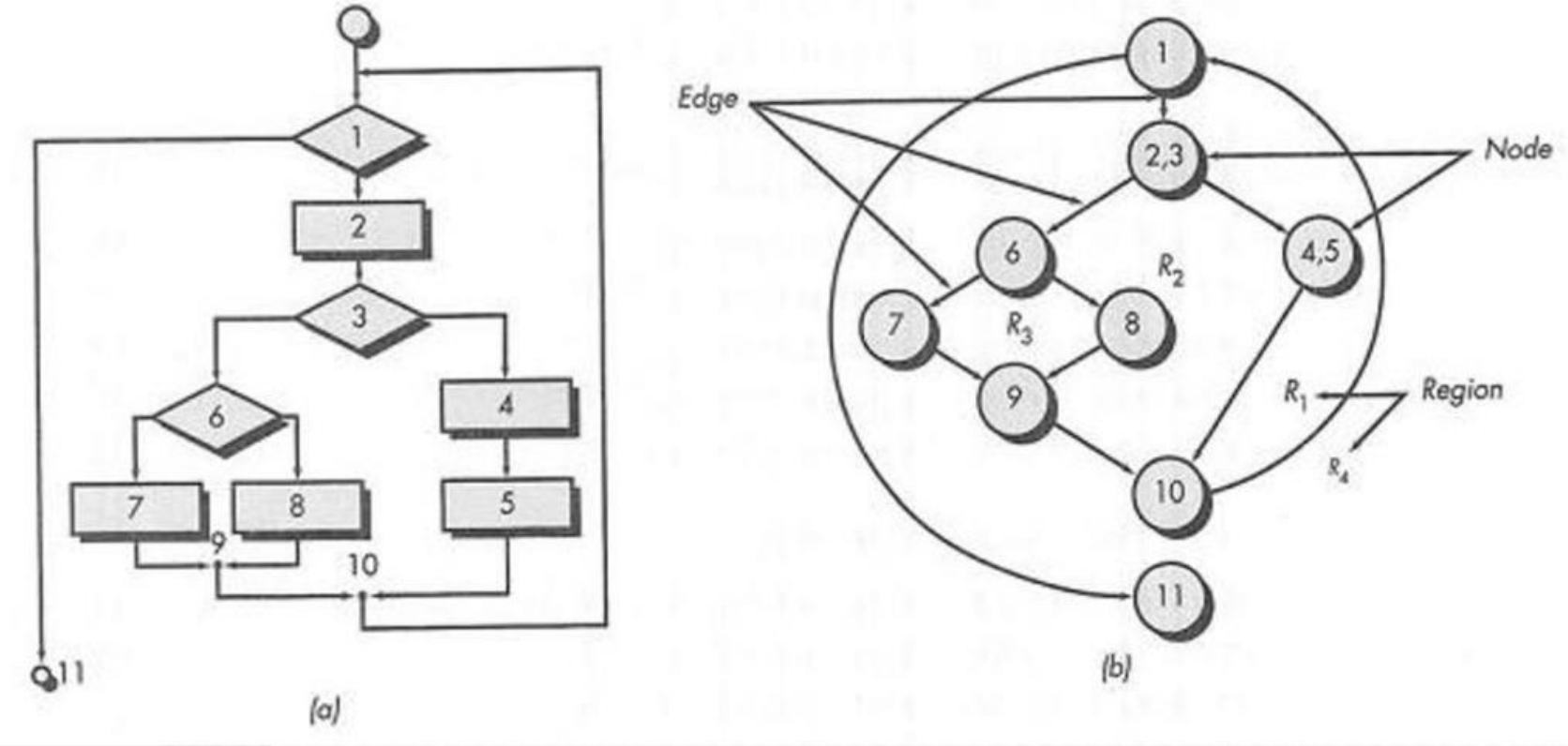
- It is a software metric that Provides a quantitative measure of the logical complexity of a program.
- Defines the number of independent paths in the basis set.
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once.
- Can be computed three ways

Cyclomatic Complexity:

- The number of regions.
- V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
- V(G) = P + 1, where P is the number of predicate nodes in the flow graph G

- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Cyclomatic Complexity:



Deriving the Basis Set and Test Cases:

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

Example:

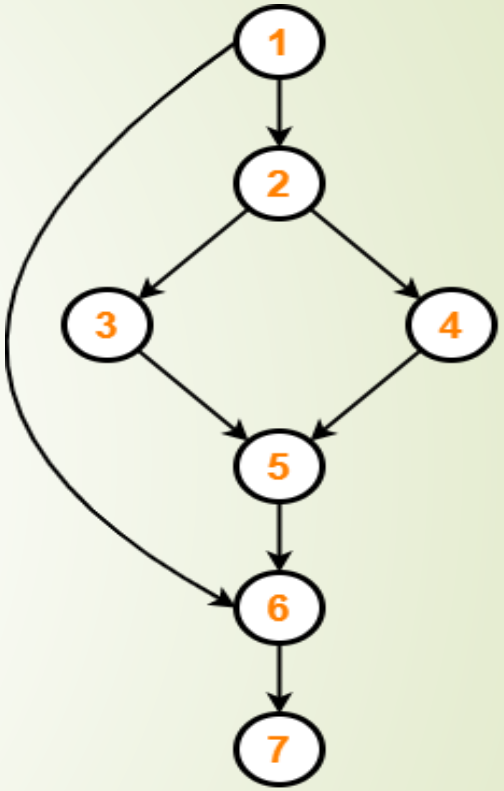
- Calculate cyclomatic complexity for the given code-

1) IF A = 354
2) THEN IF B > C
3) THEN A = B
4) ELSE A = C
5) END IF
6) END IF
7) PRINT A

Cyclomatic Complexity
= Total number of closed regions in the control flow graph
+ 1
= 2 + 1
= 3

Method-02:

Cyclomatic Complexity
= E – N + 2
= 8 – 7 + 2
= 3



Control Flow Graph

Deriving the Basis Set and Test Cases:

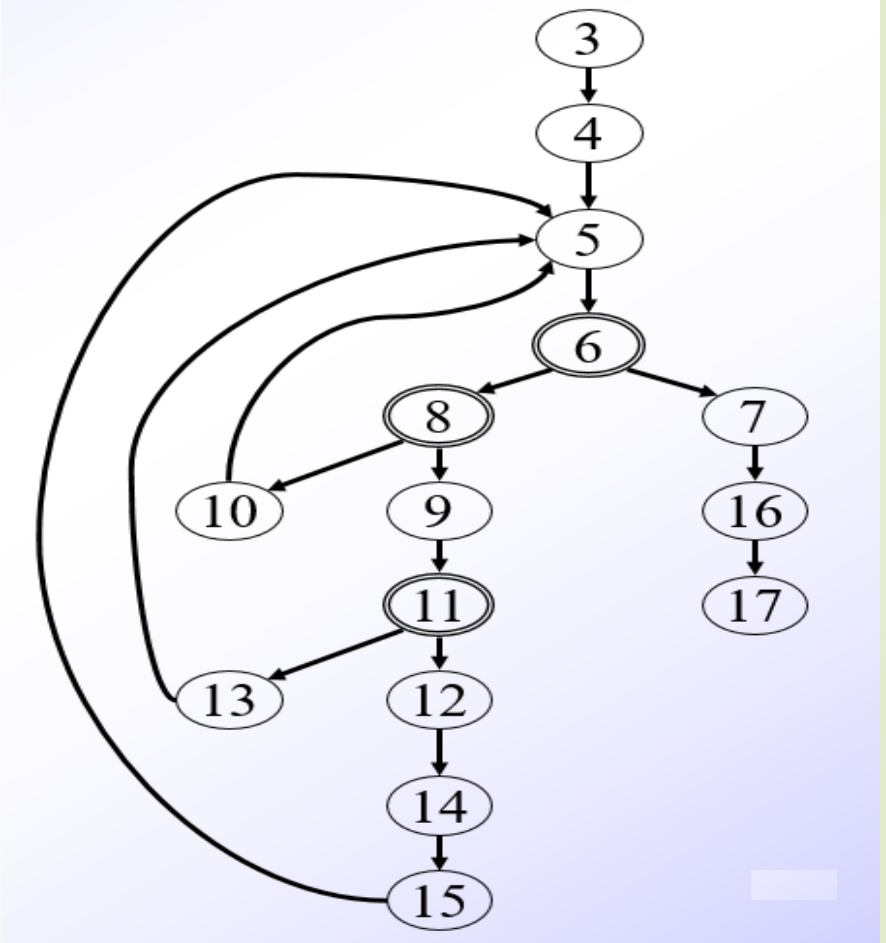
```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 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  }
```

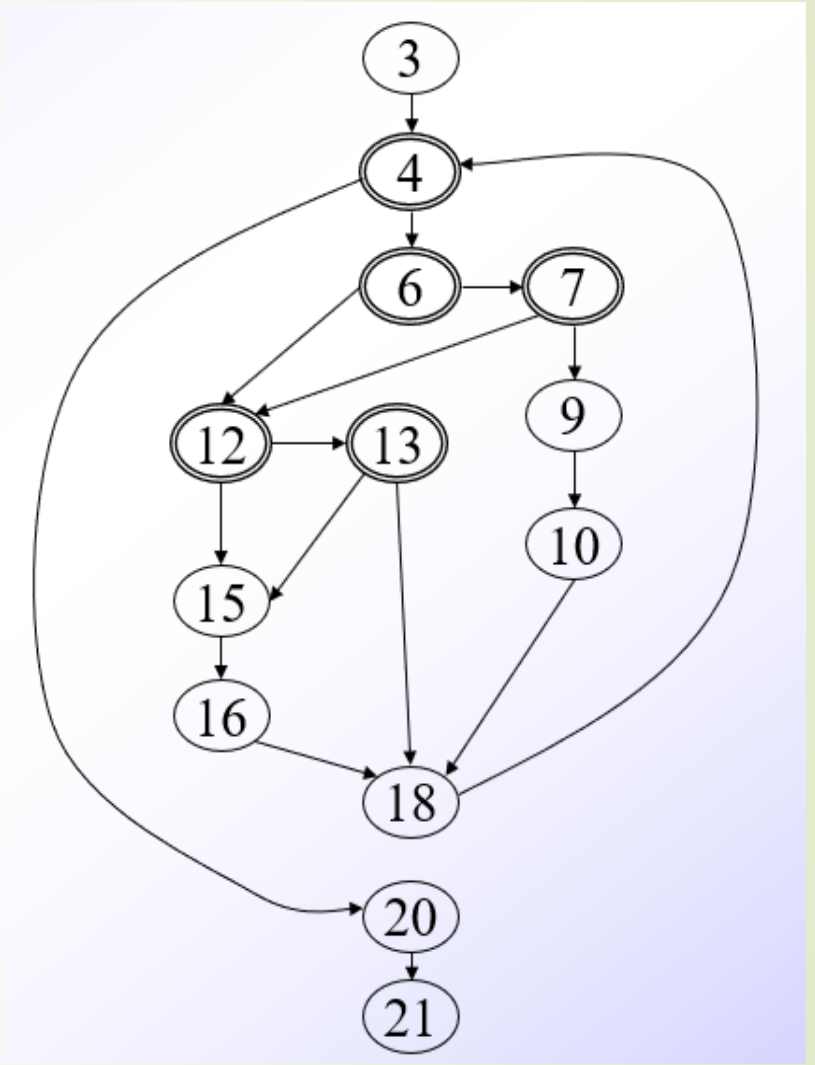


Flow Graph Example:

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7          (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13               (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

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



Focus of Glass Box Testing:

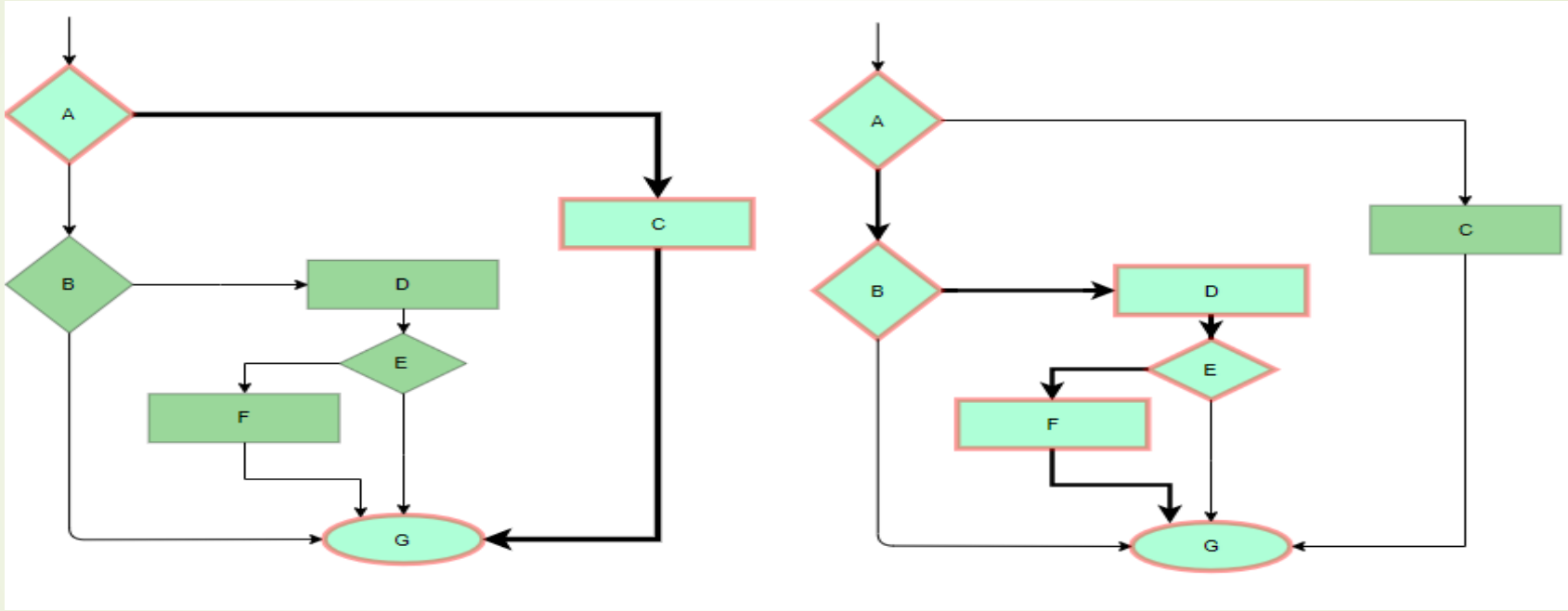
- Statement Coverage
  - Execute all statement at least ones
- Decision Coverage
  - Execute each decision direction at least once
- Condition Coverage
  - Execute each decision with all possible outcomes
- Multiple Condition Coverage
  - Invoke each point of entry at least once

Statement Coverage:

- Statement coverage is a metric to measure the percentage of statements that are executed by a test suite in a program at least once.
- Without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc.
- The aim is to traverse all statement at least once. Hence, each line of code is tested. Since all lines of code are covered, helps in pointing out faulty code.

Statement Coverage:

- In case of a flowchart, **every node** must be traversed at least once.

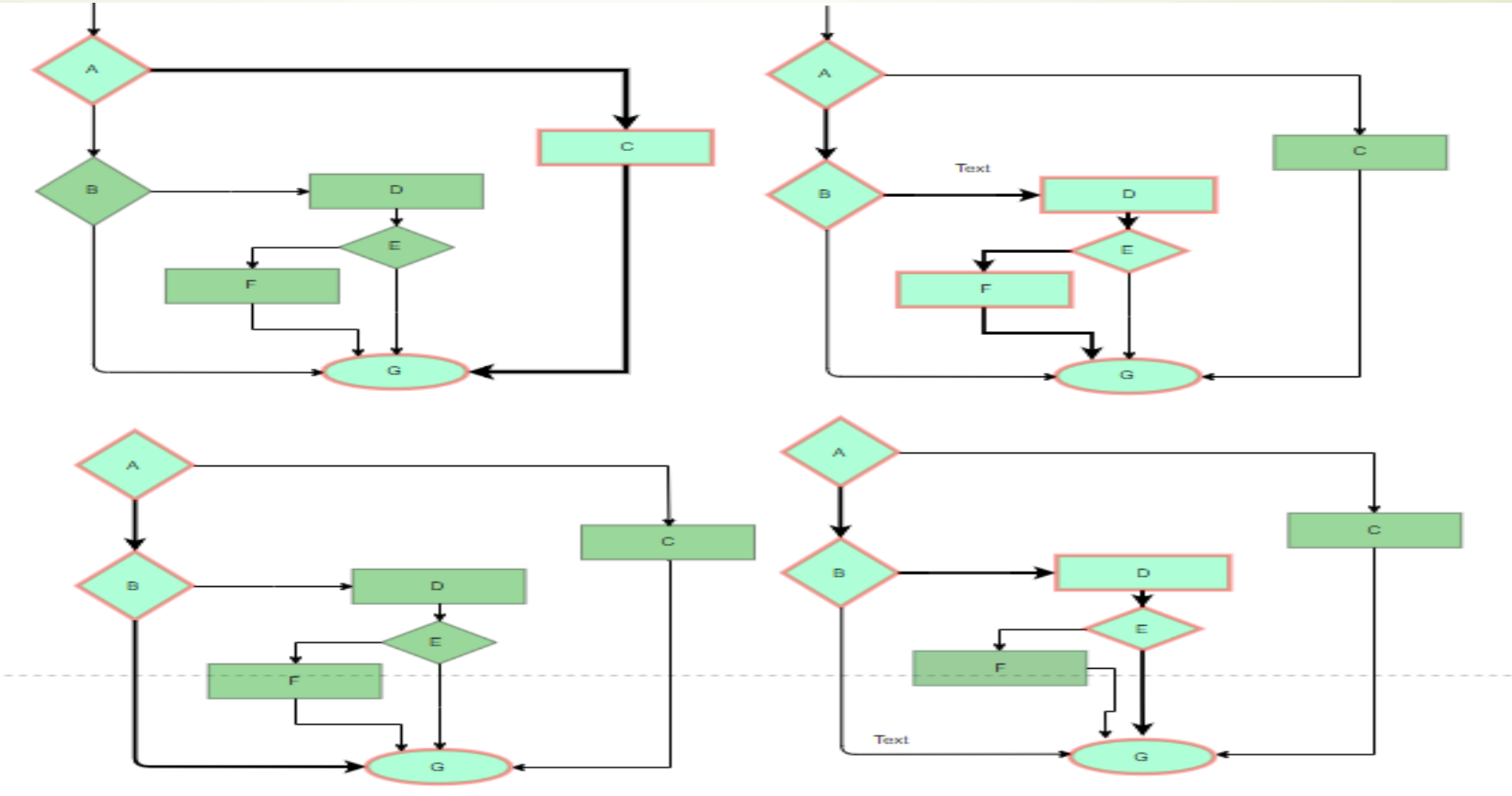


- **Weakness: Executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values.**

Branch Coverage:

- Branch coverage is also called decision coverage (DC) / edge coverage.
- A test suite achieves branch coverage, if it makes the decision expression in each branch in the program to assume both true and false values.
- Also, known as edge testing – as each edge of a program's control flow graph is required to be traversed at least once.
- In a flowchart, **all edges** must be traversed at least once.

Branch Coverage:



- **Branch coverage based testing is stronger than statement coverage based testing**
  - Branch testing would guarantee statement coverage since every statement must belong to some branch **(Assuming that there is no unreachable code).**
  - To show that statement coverage does not ensure branch coverage- **give an example of a test suite that achieve statement coverage, but does not cover at least one branch.**
  - **If (x>2) x+=1; and the test suite {5};**

Condition Coverage:

- Condition coverage testing is also known as basis condition testing.
- A test suite is said to achieve basic condition coverage (BCC), if each basic condition in every conditional expression assumes both true and false values during testing.
- Example: if (A || B && C); the basic conditions A, B, and C assume both true and false values.
  - Here just two test cases can achieve condition coverage {A=True, B=True, and C=True} {A=False, B=False, and C=False}

Condition Coverage:

- Condition Coverage: In this technique, all individual conditions must be covered as shown in the following example:
  - **READ X, Y**
  - **IF(X == 0 || Y == 0)**
  - **PRINT '0'**
  - In this example, there are 2 conditions: X == 0 and Y == 0. Now, test these conditions get TRUE and FALSE as their values. One possible example would be:
  - **#TC1 – X = 0, Y = 5**
  - **#TC2 – X = 5, Y = 0**
- **Condition coverage may not achieve branch coverage.**

Multiple Condition Coverage:

- Multiple condition coverage (MCC) is achieved, if the test cases make the component conditions of a composite conditional expression to assume all possible combinations true and false values.
- Example: Consider the composite conditional expression [(c1 and c2) or C3].
- A test suite would achieve MCC, if all the component conditions c1, c2, and c3 are each made to assume all combinations of true and false values.

Multiple Condition Coverage:

- Let's consider the following example:
- READ X, Y
- IF(X == 0 || Y == 0)
- PRINT '0'
- #TC1: X = 0, Y = 0
- #TC2: X = 0, Y = 5
- #TC3: X = 55, Y = 0
- #TC4: X = 55, Y = 5
- Hence, four test cases required for two individual conditions. Similarly, if there are n conditions then 2^n test cases would be required.

- **Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.**

```
If (temperature >150 | | temperature>50)
    setWarningLightON();
```

- The program segment has a bug in the second component condition, it should have been temperature < 50.
- The test suite {temperature =160, temperature =40} achieves branch coverage. But, it is not able to check that SetWarningLightON(); should not be called for temperature values withing 150 and 50.

Black-box Testing

Thank You.