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

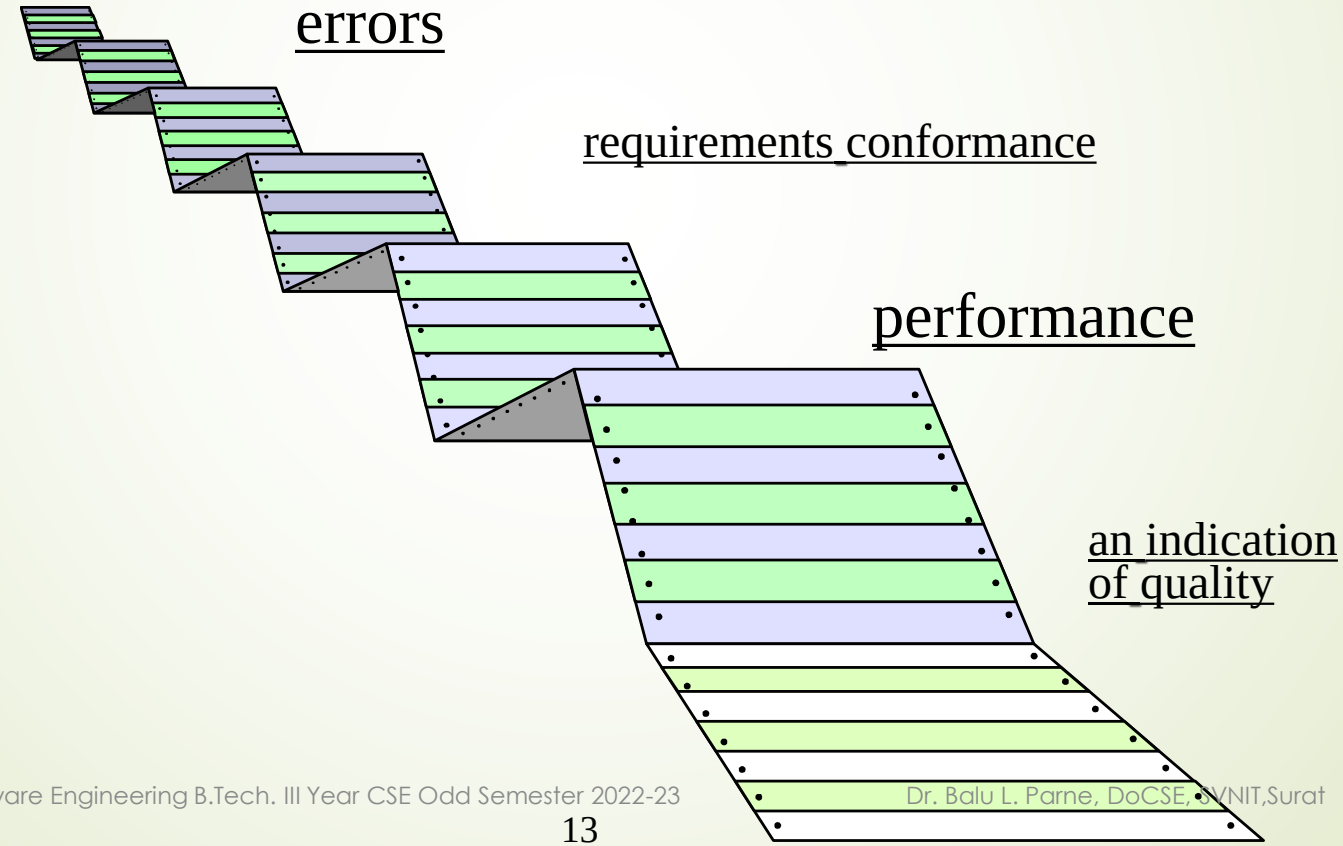
- **Operability**
  - The better it works (i.e., better quality), the easier it is to test
- **Observability**
  - Incorrect output is easily identified; internal errors are automatically detected
- **Controllability**
  - The states and variables of the software can be controlled directly by the tester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

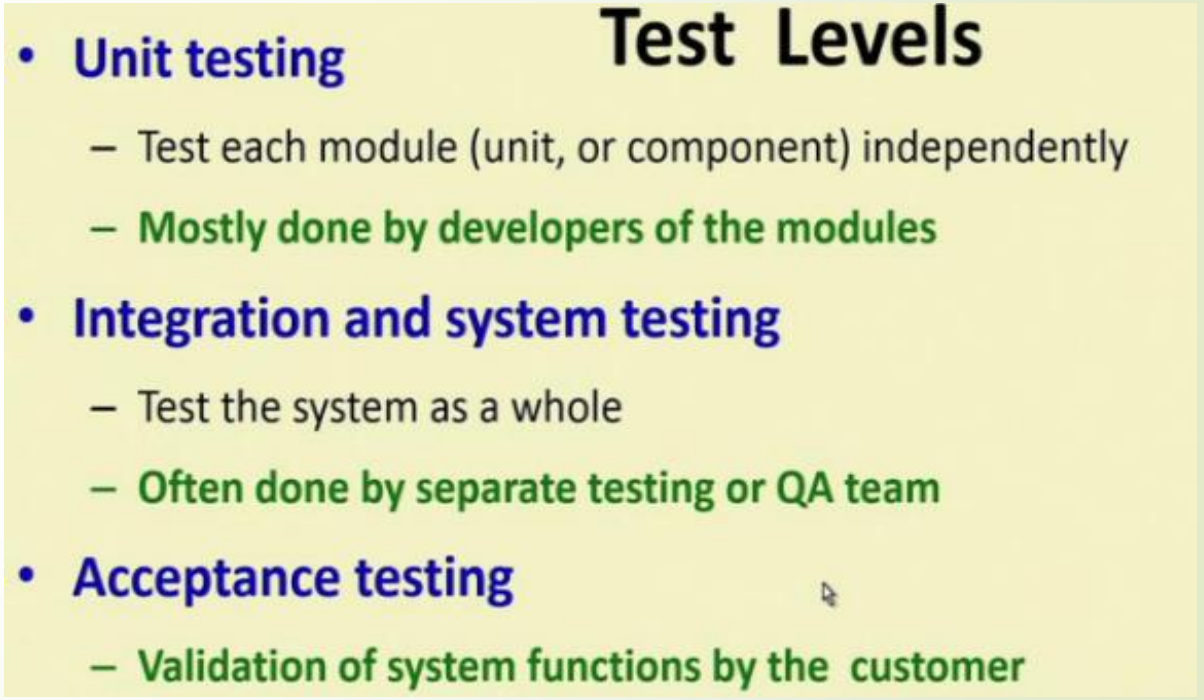
Verification and Validation Techniques

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

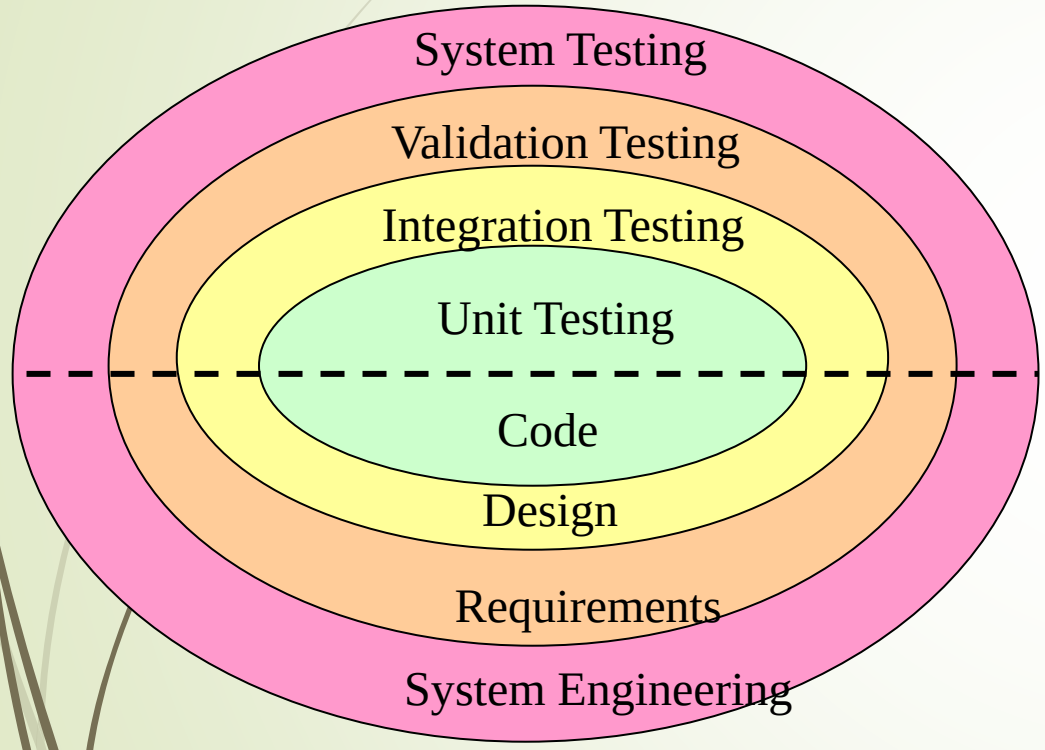
Verification	Validation
Are you building it right?	Have you built the right thing?
Checks whether an artifact conforms to its previous artifact.	Checks the final product against the specification.
Done by developers.	Done by Testers.
Static and dynamic activities: reviews, unit testing.	Dynamic activities: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

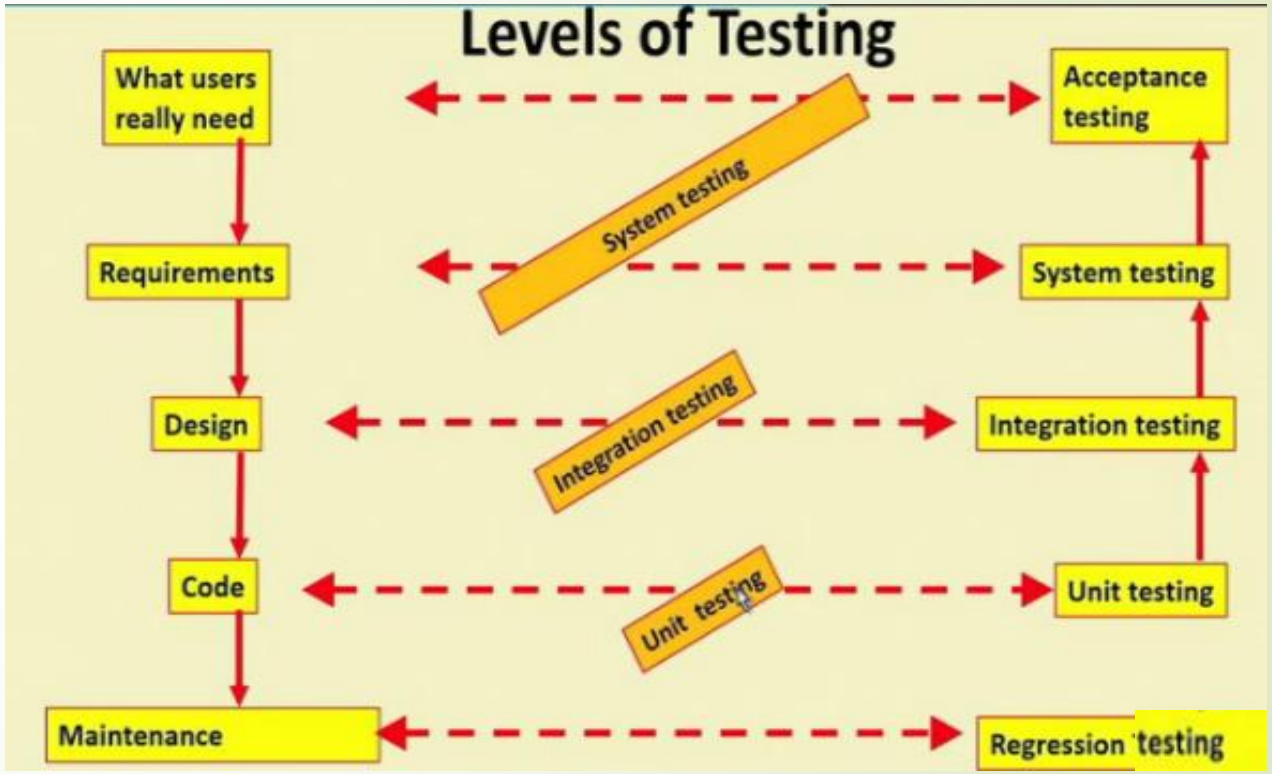
4 Testing Levels:

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

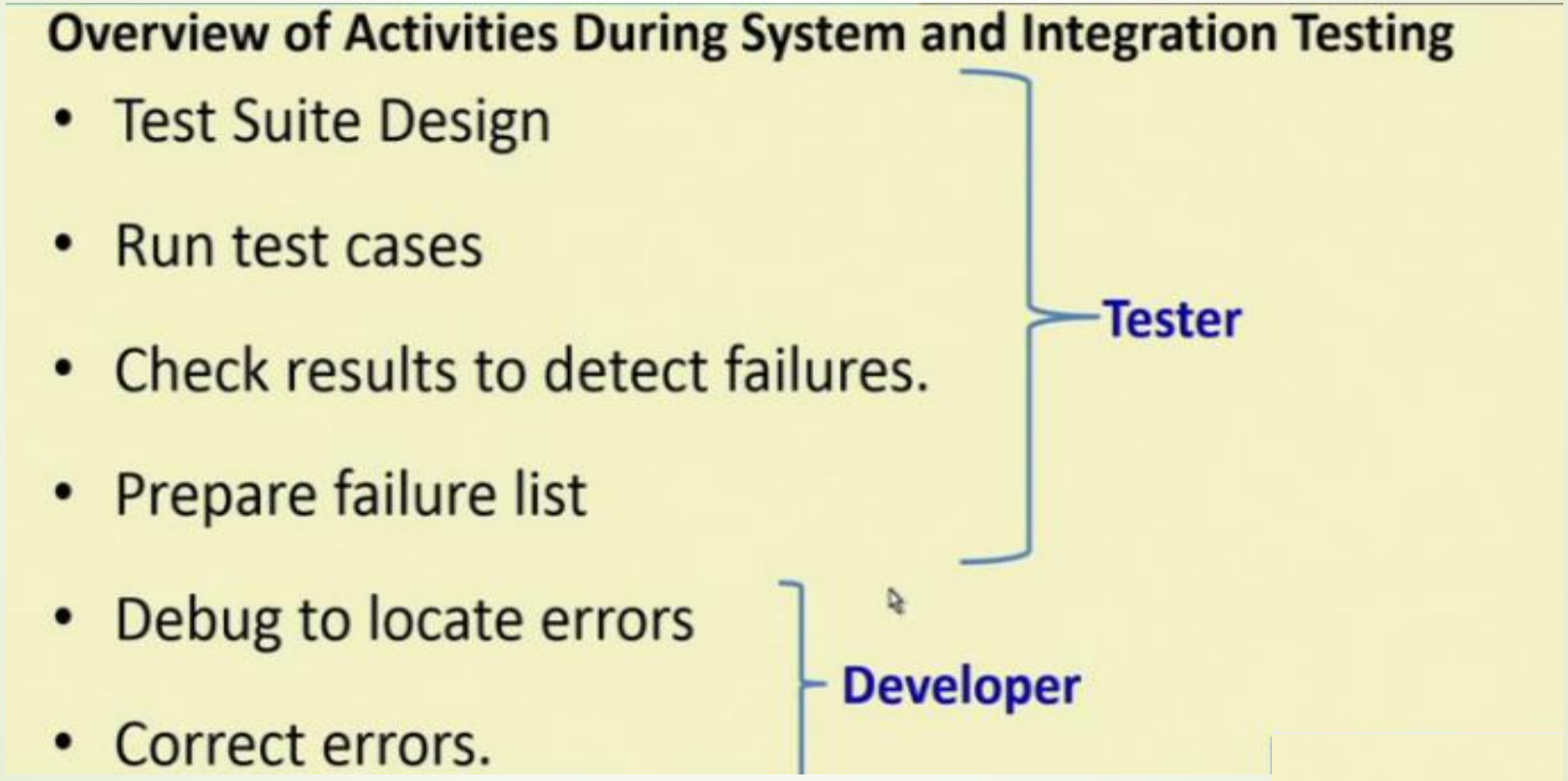


Levels of Testing:





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**

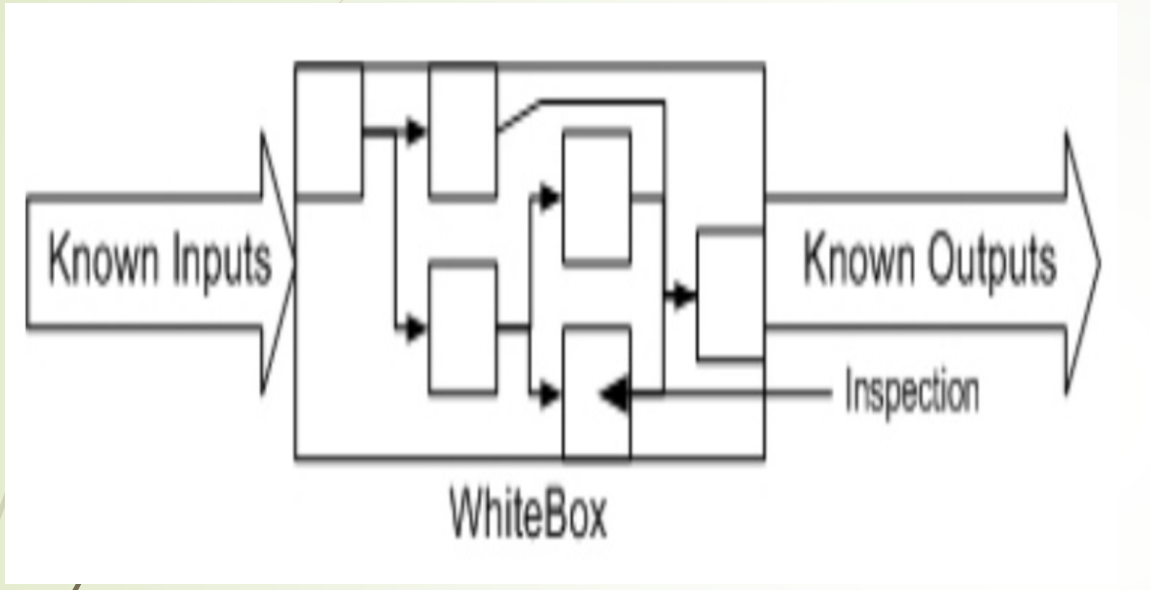
Design of Test Cases:

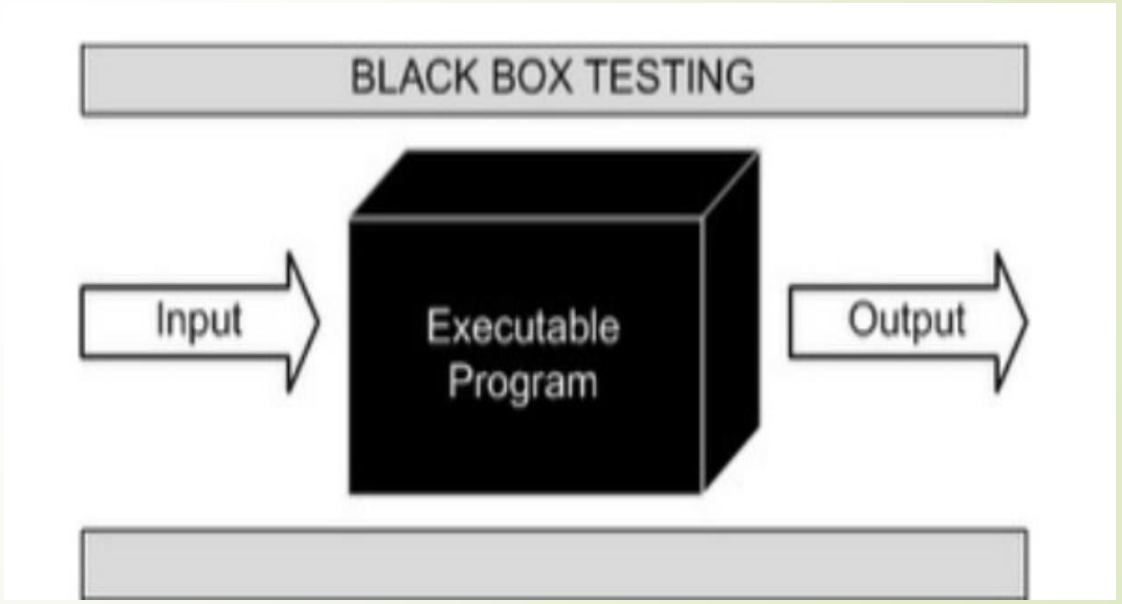
- Testing a system using a large number of randomly selected test cases:
  - Does not mean that most errors in the system will be uncovered.
- Consider the function that find maximum of two integers x and y.

```
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

White-box and Black-box Testing:





White-box Testing:

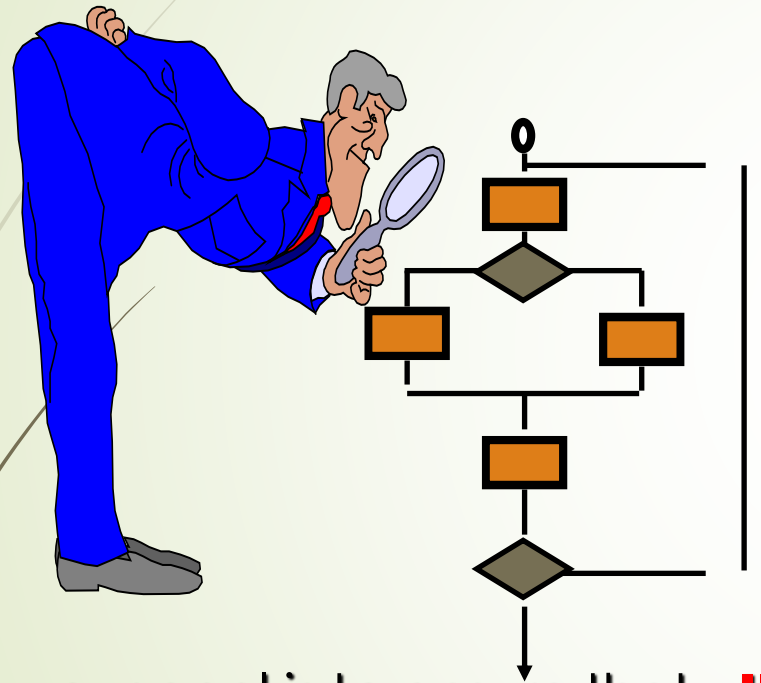
- Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised.
- **Involves tests that concentrate on close examination of procedural detail.**
- **Logical paths through the software are tested.**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

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.**

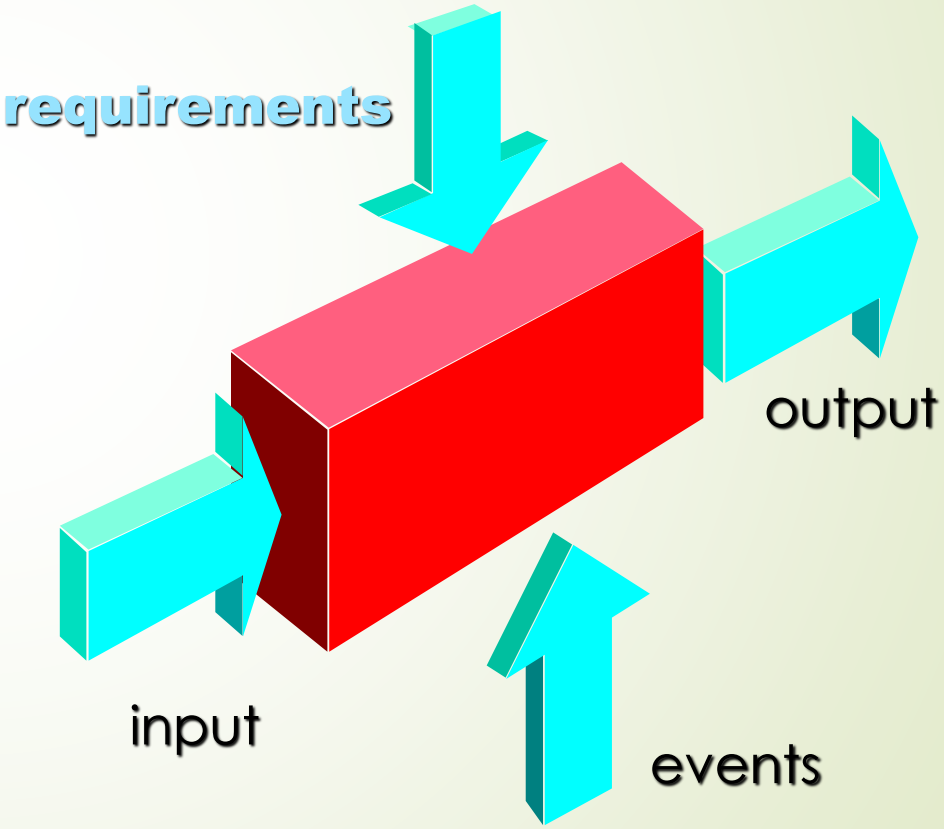
Software Testing

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.

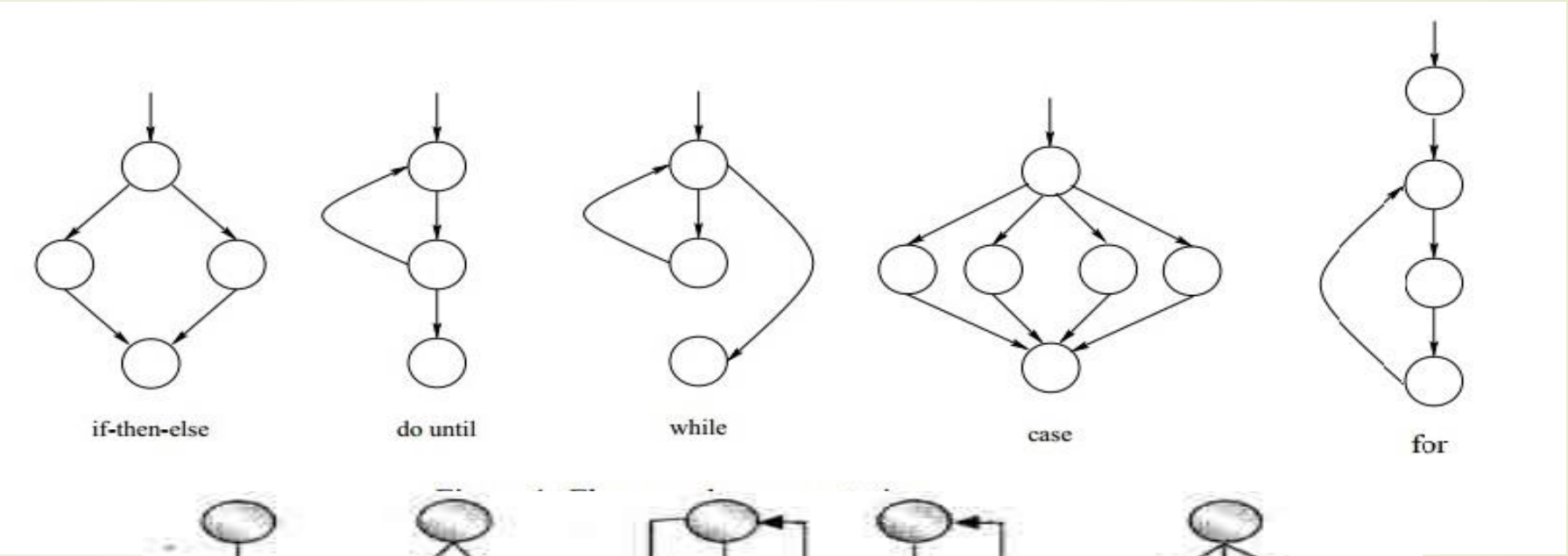
Flow Graph Notation:

- 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:

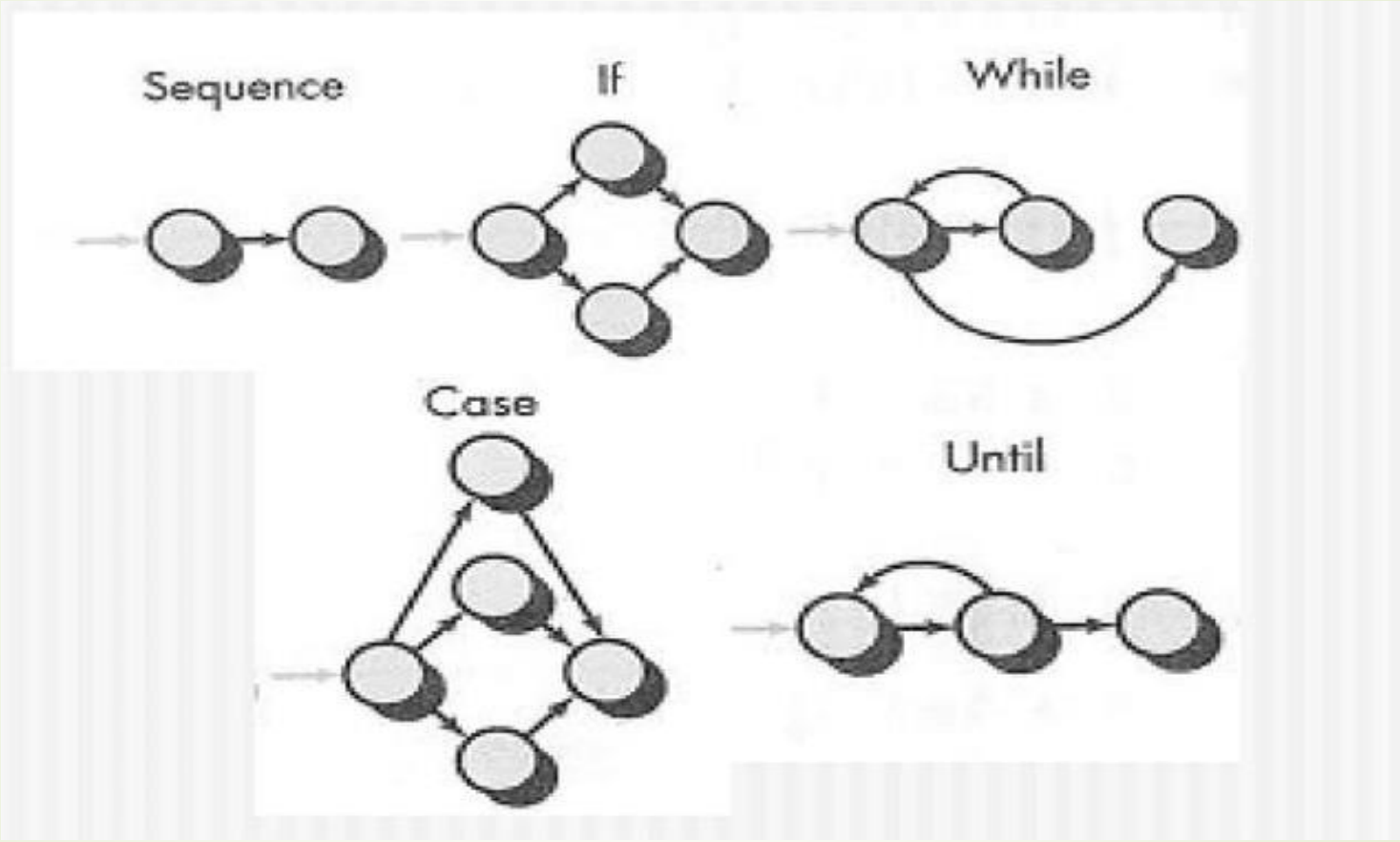
- The arrows on the flow graph, called edges or links , represents flow of control and are analogous to flowchart arrow.
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions.
- When counting regions, include the area outside the graph as a region.

Flow Graph Notation:



Flow Graph Notation

Flow Graph Notation:

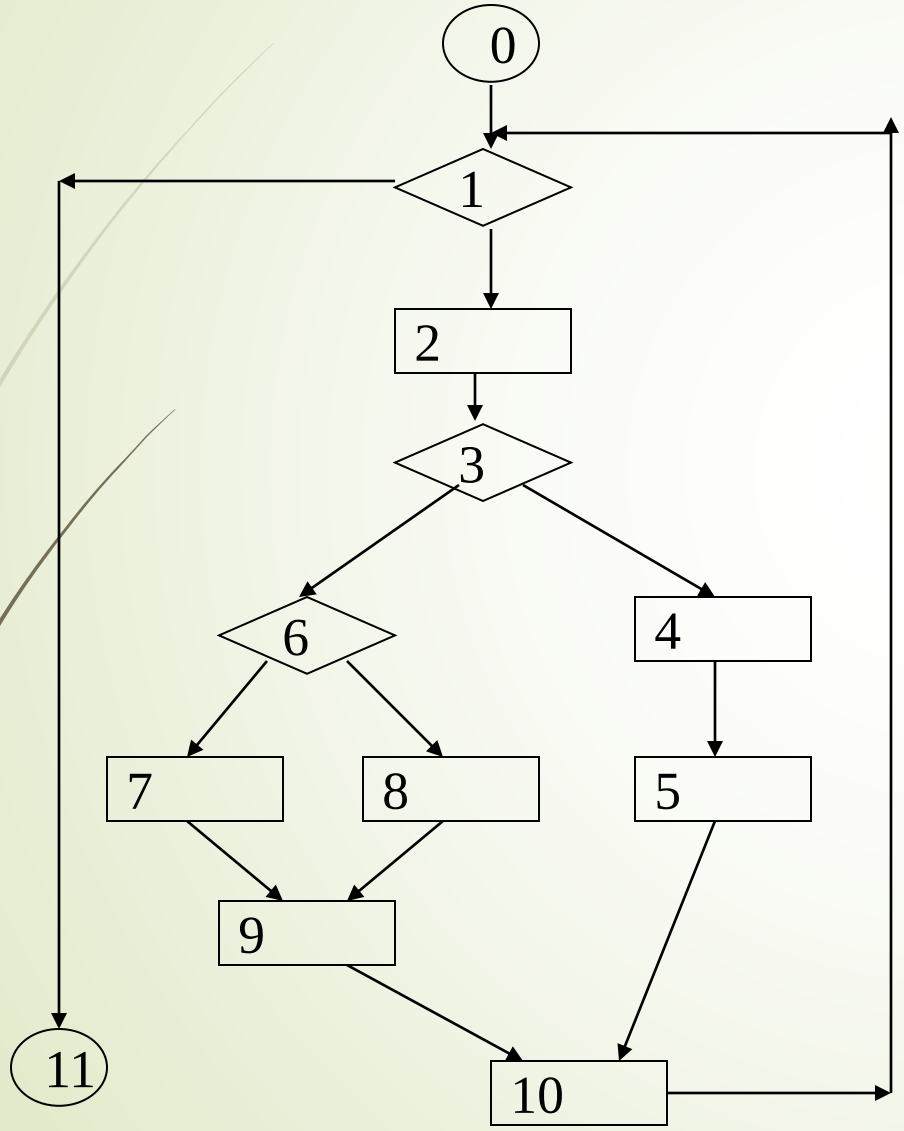


Flow Graph Notation:

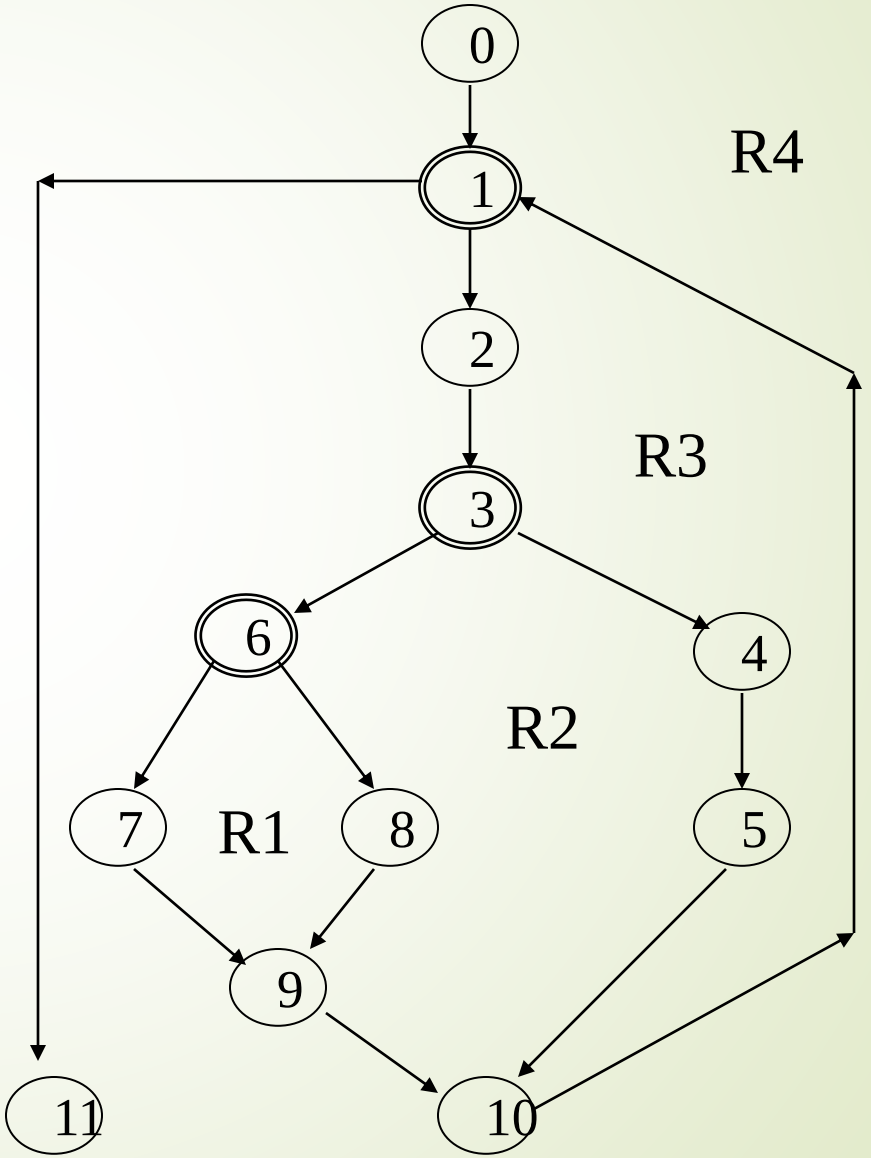
- A node containing a simple conditional expression is referred to as a predicate node.
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True 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

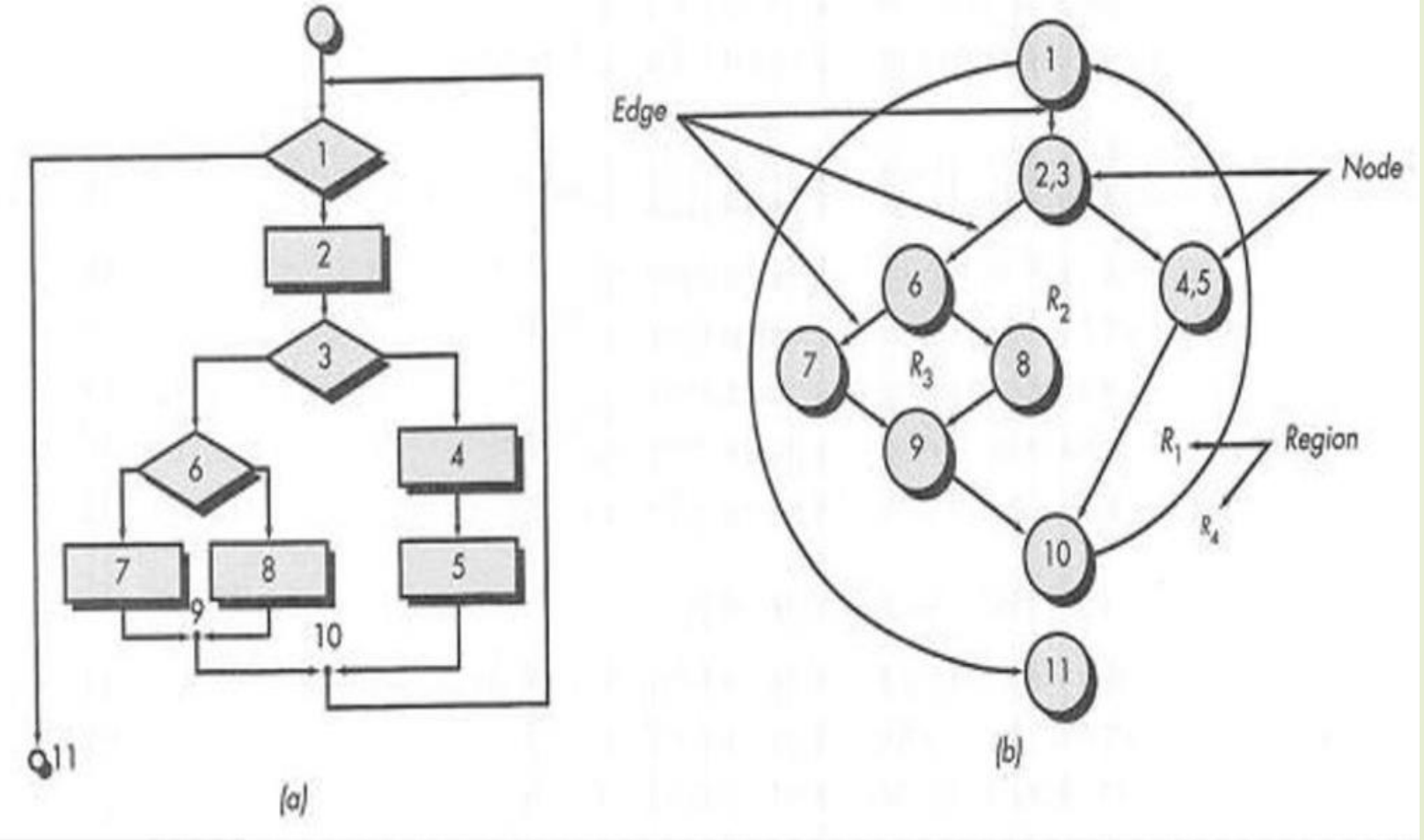
FLOW CHART



FLOW GRAPH



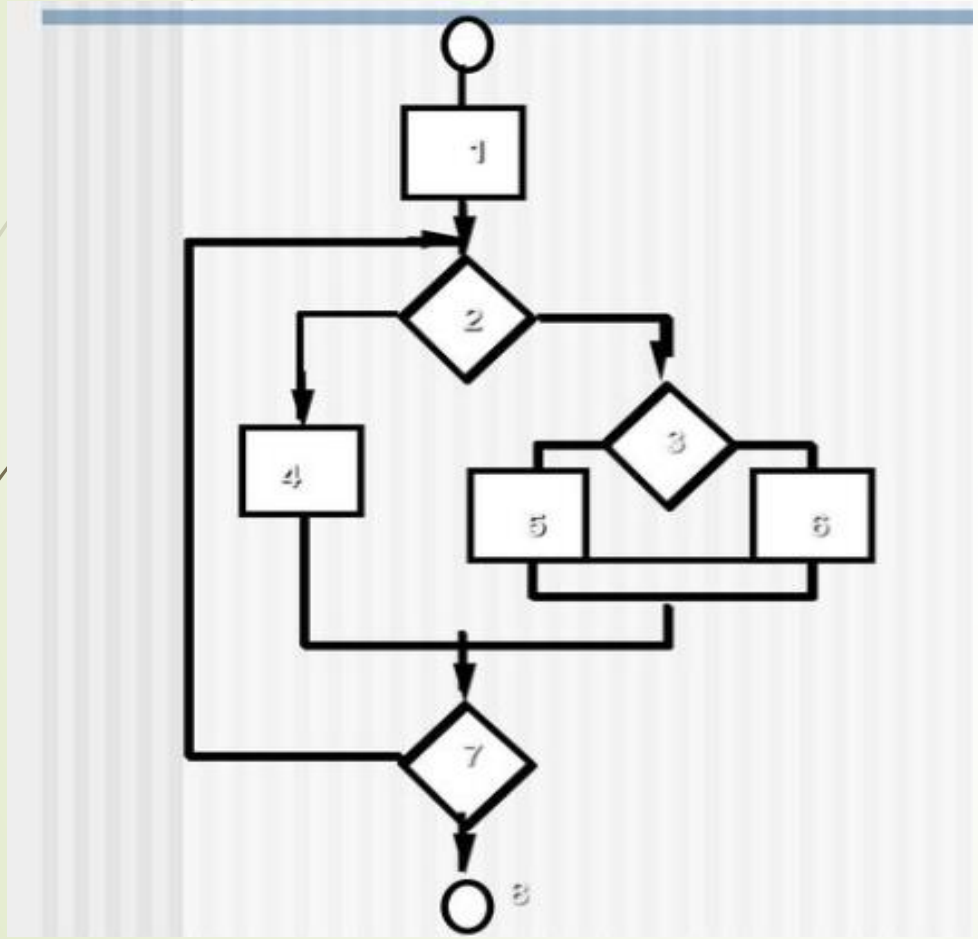
Flow chart and Flow Graph:



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:

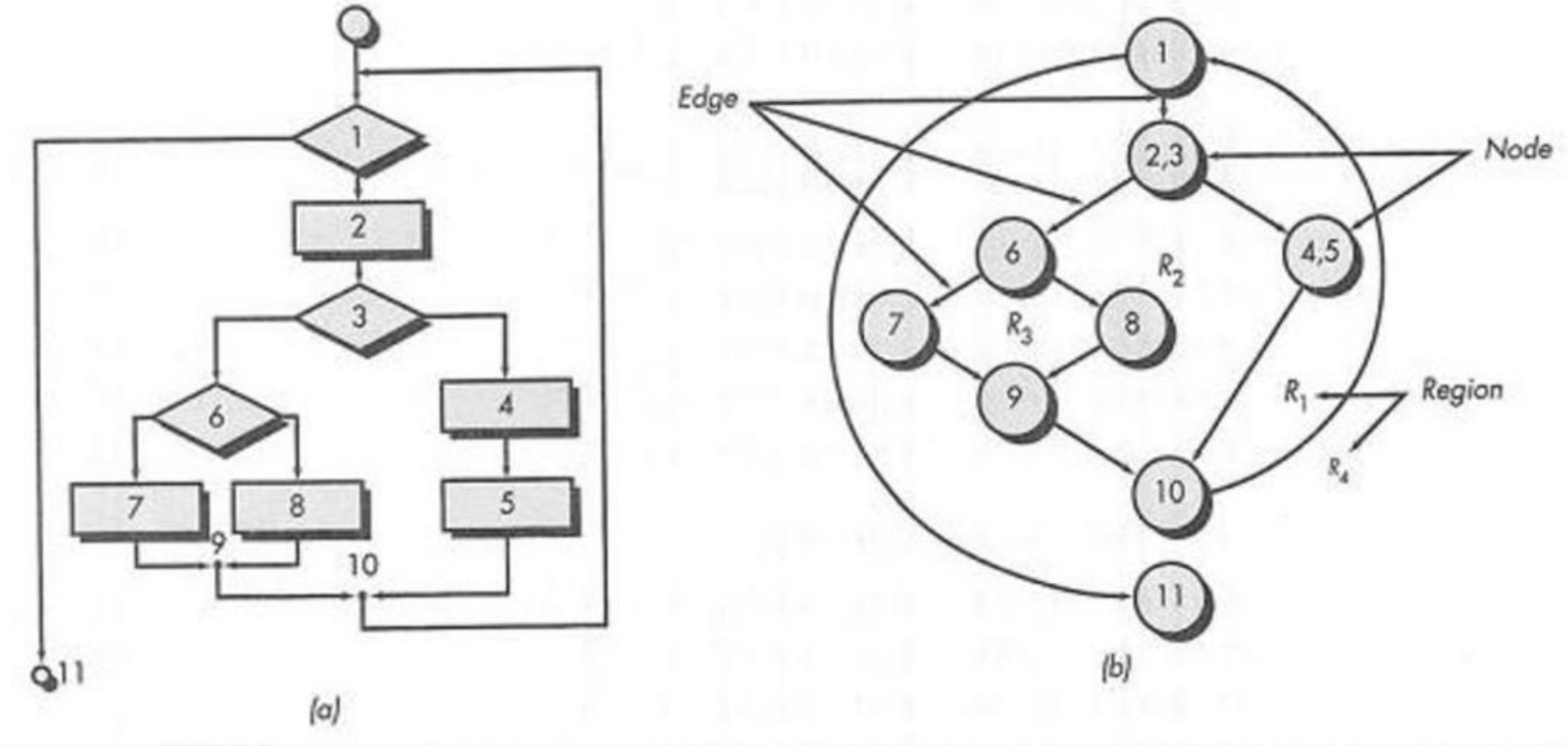
- It is a software metric that Provides a quantitative measure of the logical complexity of a program.
- Defines the number of independent paths in the basis set.
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once.
- Can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

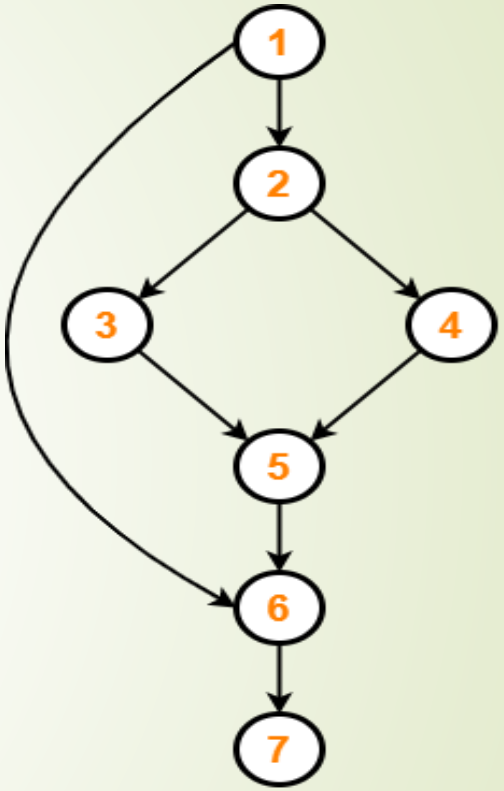
- Calculate cyclomatic complexity for the given code-

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:

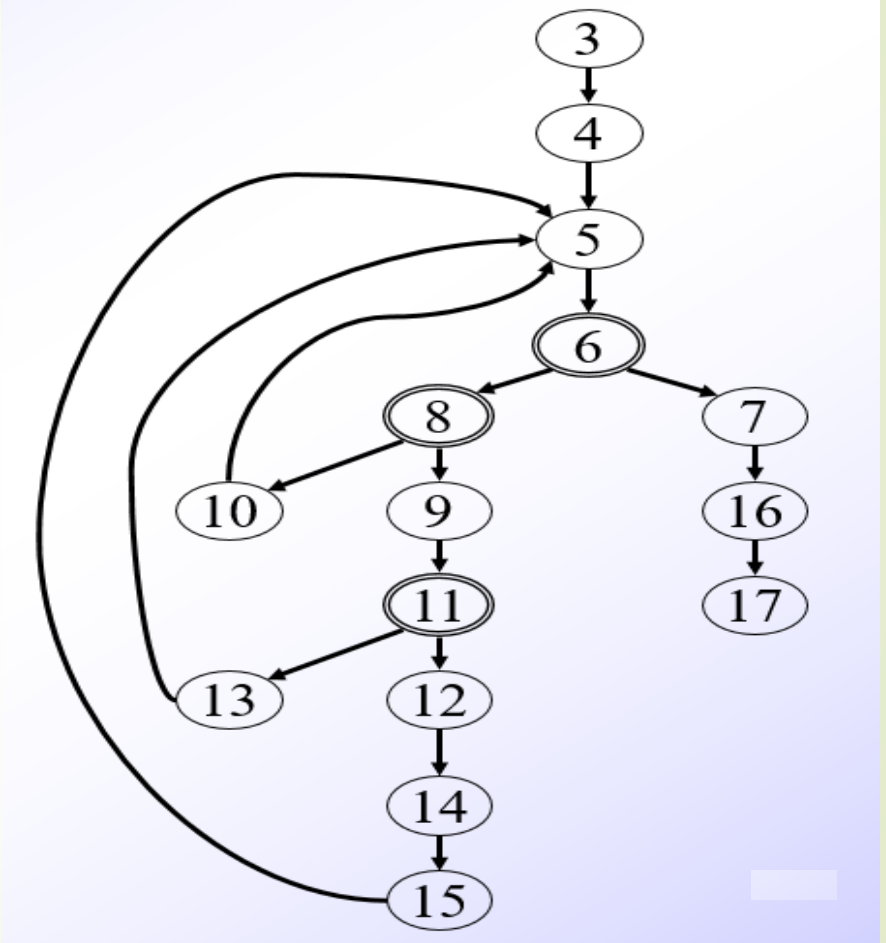
```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 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  }
```

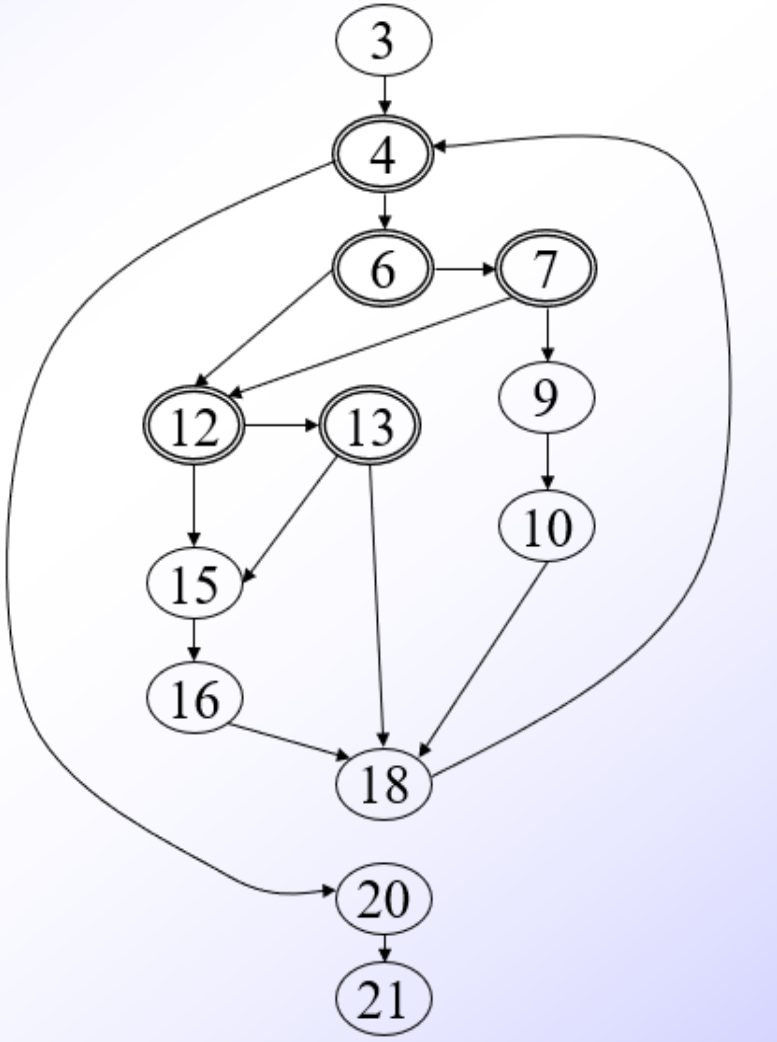


Flow Graph Example:

```
 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:

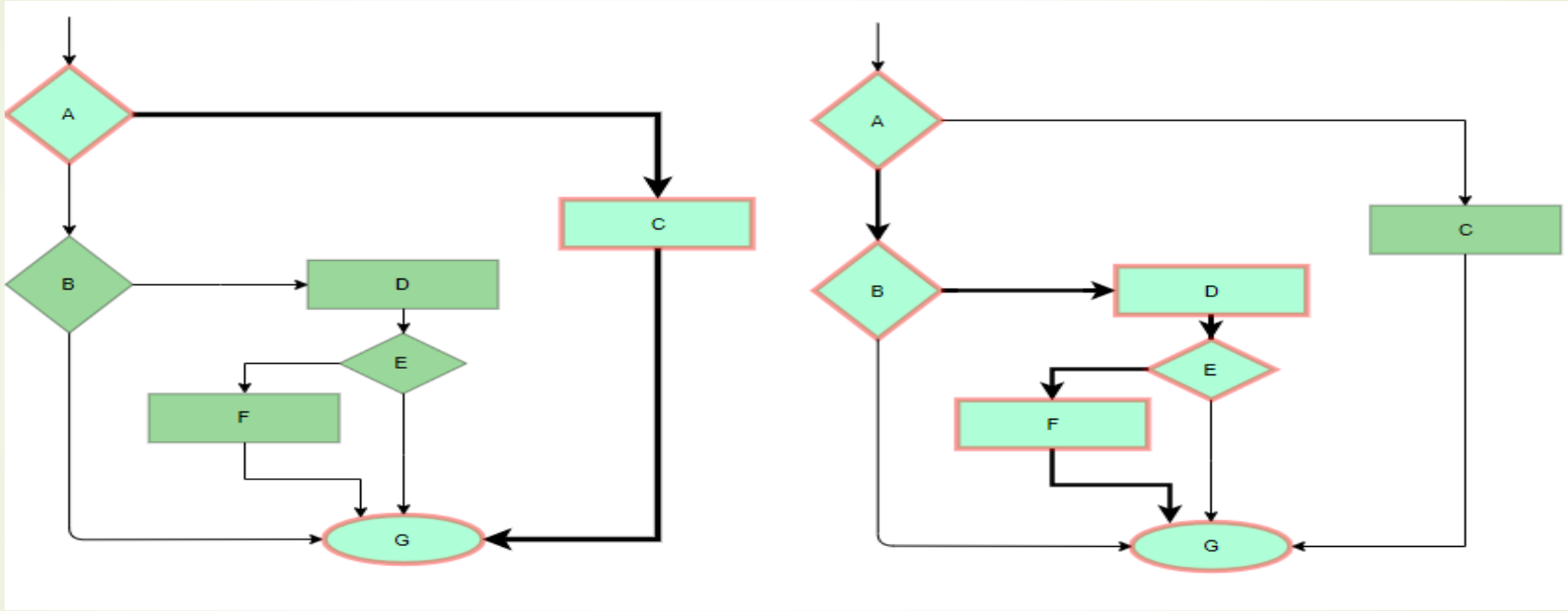
- Statement Coverage
  - Execute all statement at least ones
- Decision Coverage
  - Execute each decision direction at least once
- Condition Coverage
  - Execute each decision with all possible outcomes
- Multiple Condition Coverage
  - Invoke each point of entry at least once

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.

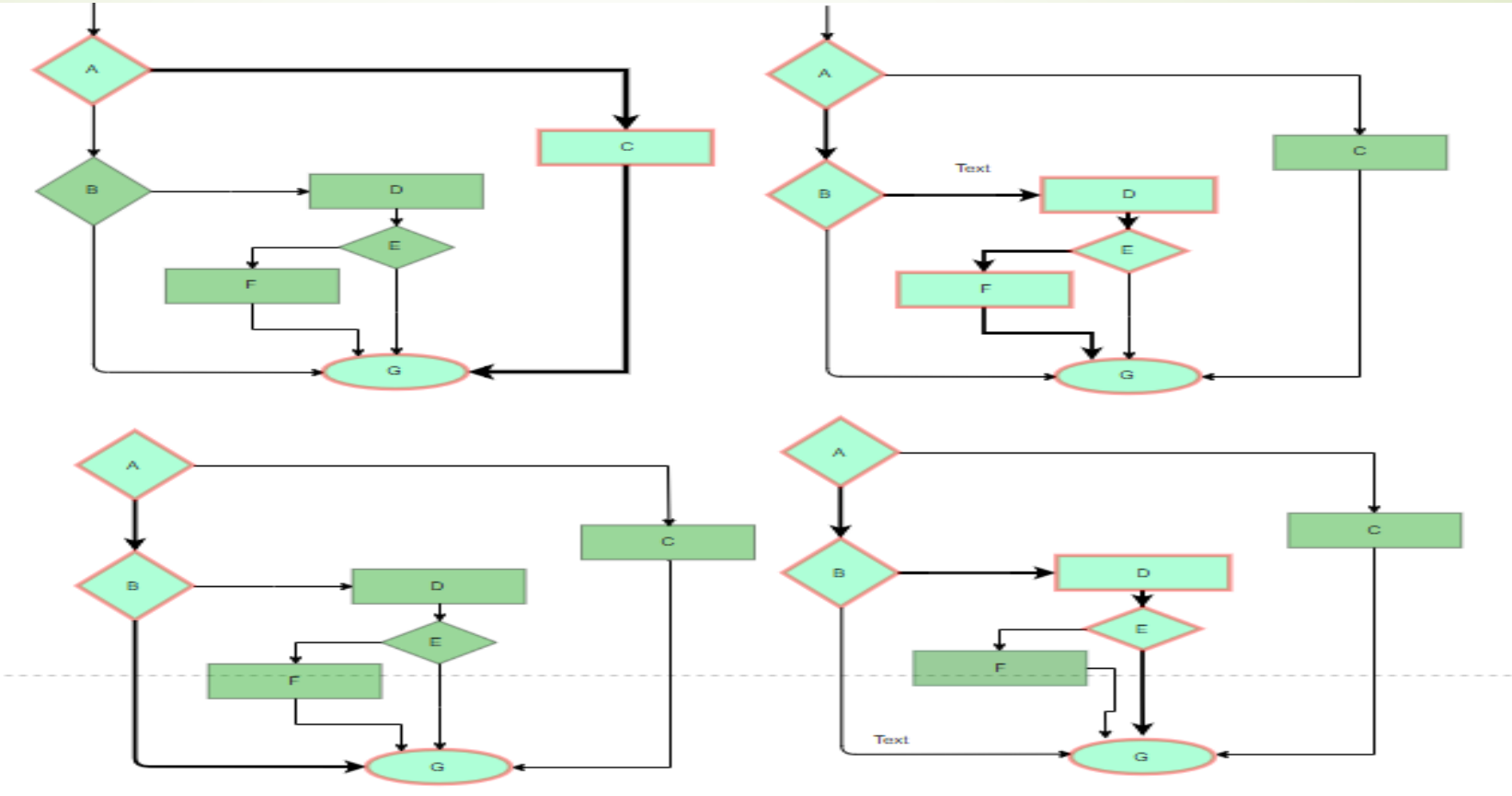


- **Weakness: Executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values.**

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

Black-box Testing:

- **Complements white-box testing** by uncovering different classes of errors.
- Also called as **Behavioral testing**, Focuses on the functional requirements and the information domain of the software.
- Used during the later stages of testing after white box testing has been performed.
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program.
- Attempts to find errors in the following categories:
  - Incorrect or missing functions
  - Interface errors
  - Behavioral or performance errors
  - Initialization and termination errors.
  - Errors in data structure or external data base access.

Questions answered by Black-box Testing:

- How is **functional validity** tested**?**
- How are **system behavior and performance** tested**?**
- What **classes of input** will make good test cases**?**
- Is the system particularly **sensitive to certain input values?**
- How are the **boundary values of a data** class isolated**?**
- What **data rates and data volume** can the system tolerate**?**
- What effect will **specific combinations of data** have on system operation**?**

Black Box Testing Techniques:

- Equivalence Partitioning
- Boundary Value Analysis
- Error Guessing
- Cause Effect Graphing

Equivalence Partitioning:

- A black-box testing method that **divides the input domain of a program into classes of data** from which test cases are derived.
- An ideal test case single-handedly uncovers a complete class of errors, thereby **reducing the total number of test cases that must be developed.**
- Test case design is based on an evaluation of equivalence classes for an input condition.
- An equivalence class represents a set of valid or invalid states for input conditions.
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once.

Equivalence Partitioning:

- Equivalence partitioning – **It is often seen that many type of inputs work similarly** so instead of giving all of them separately **we can group them together and test only one input of each group.**
- The idea is to **partition the input domain of the system into a number of equivalence** classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error.
- Identification of equivalence class – **Partition any input domain into minimum two sets: valid values and invalid values.** For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.

Guidelines for Defining Equivalence Classes

- If an **input condition specifies a range,** one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an **input condition requires a specific value,** one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an **input condition specifies a member of a set,** one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an **input condition is a Boolean value,** one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

Guidelines for Defining Equivalence Classes

- **Generating test cases –**
  - To each valid and invalid class of input assign unique identification number.
  - Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.
- Example:
- A program which edits credit limits within a given range ($10,000 - $15,000) would have three equivalence classes
  - **<**$10,000(invalid)
  - Between $10,000 & $15,000(valid)
  - **>**$15,000(invalid)

Boundary Value Analysis:

- A greater number of errors occur at the boundaries of the input domain rather than in the "center".
- Boundaries are very good places for errors to occur. Hence, **if test cases are designed for boundary values of input domain then the efficiency of testing improves and probability of finding errors also increase.**
- Boundary value analysis is a test case design method that **complements equivalence partitioning**
  - It selects test cases at the edges of a class.
  - It derives test cases from both the input domain and output domain.

Guidelines for Boundary Value Analysis:

- If **an input condition specifies a range** bounded by values a and b, test cases should be designed with values a and b as well as values just above and just below a and b.
- If **an input condition specifies a number of values,** test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested.
- Apply above guidelines to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above.
- If **internal program data structures have prescribed boundaries** (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries.

Boundary Value Analysis:

- This technique consist of building test cases and data that focus on the input and the output boundaries of a given function.
- Example:
- A program which edits credit limits within a given range ($10,000 - $15,000) would have three equivalence classes
  - Low Boundary +/- one($9,999 & $10,001)
  - On the Boundary ($10,000 & $15,000)
  - Upper Boundary -/+ one($14,999 & $15,001)

Error Guessing:

- Error Guessing is a Software Testing technique on **guessing the error** which can prevail in the code.
- Test cases can be developed based upon **the intuition and experience of the tester.**
- Example:
- Where one of the **inputs is the date**, a tester may **try February 29, 2001.**

Cause-Effect Graphing:

- Cause Effect Graph is a black box testing technique that **graphically illustrates the relationship** between a given outcome and all the factors that influence the outcome.
- Cause Effect Graph is a technique in which a **graph is used to represent the situations of combinations of input conditions.**
- Cause-effect graphing technique is used because boundary value analysis and equivalence class partitioning methods do not consider the **combinations of input conditions.**
- There may be some critical behavior to be tested when some combinations of input conditions are considered, that is why cause-effect graphing technique is used.

Cause-Effect Graphing:

- Cause Effect Graph is used **to write dynamic test cases.**
- The dynamic test cases are used when code works dynamically based on user input.
  - For example, while using email account, on entering valid email, the system accepts it but, when you enter invalid email, it throws an error message.
- In this technique, the input conditions are assigned with causes and the result of these input conditions with effects.
- Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

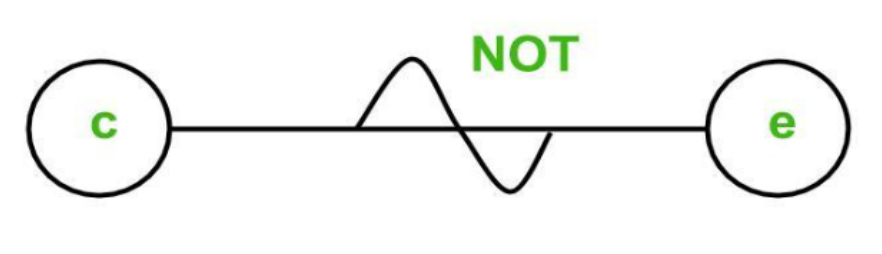
Cause-Effect Graphing:

- The main advantage of cause-effect graph testing is, it reduces the time of test execution and cost.
- This technique aims to reduce the number of test cases but still covers all necessary test cases with maximum coverage to achieve the desired application quality.
- Cause-Effect graph technique converts the requirements specification into a logical relationship between the input and output conditions by using logical operators like AND, OR and NOT.

Basic Notations used in Cause-effect graph:

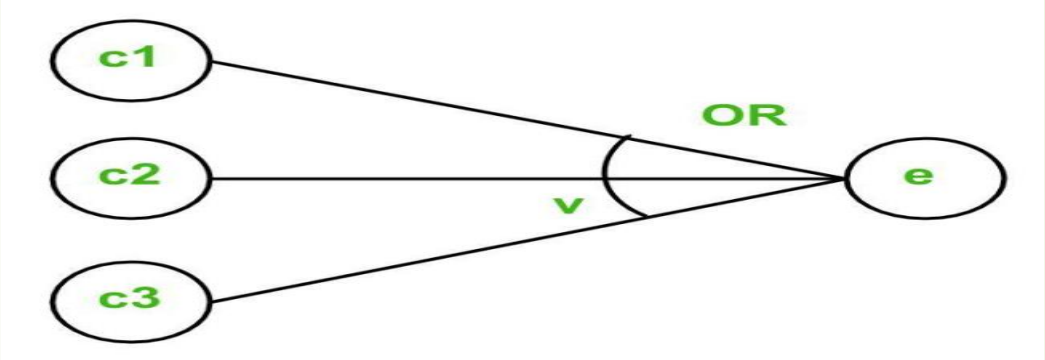
- **Identity Function:** if c is 1, then e is 1. Else e is 0.

- **NOT Function:** if c is 1, then e is 0. Else e is 1.

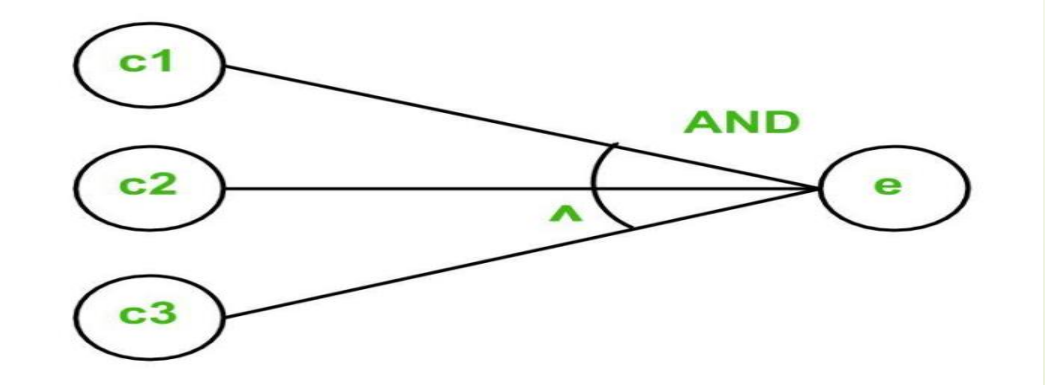


Basic Notations used in Cause-effect graph:

- **OR Function:** if c1 or c2 or c3 is 1, then e is 1. Else e is 0.

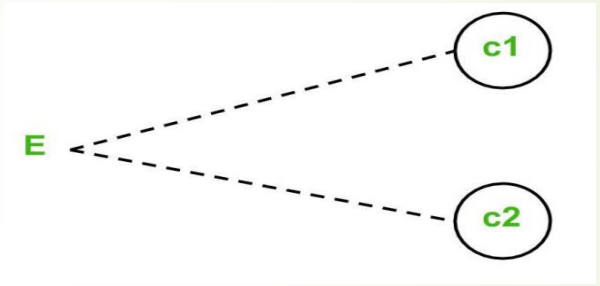


- **AND Function:** if both c1 and c2 and c3 is 1, then e is 1. Else e is 0.

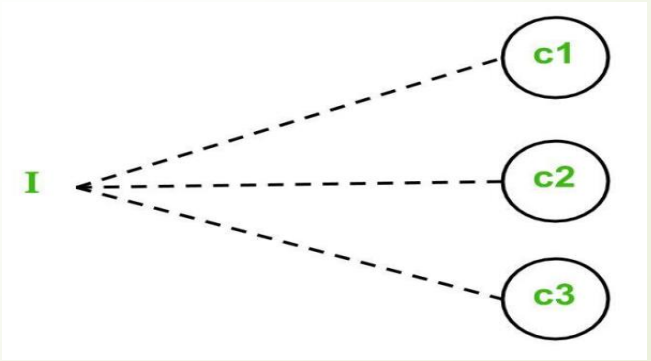


Constraints in Cause-effect graph:

- **Exclusive constraint or E-constraint:** This constraint exists between causes. It states that either c1 or c2 can be 1, i.e., c1 and c2 cannot be 1 simultaneously.

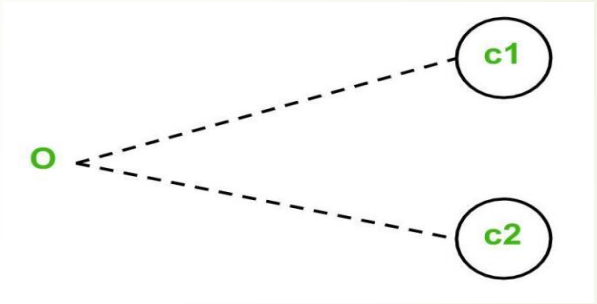


- **Inclusive constraint or I-constraint:** This constraint exists between causes. It states that at least one of c1, c2 and c3 must always be 1, i.e., c1, c2 and c3 cannot be 0 simultaneously.

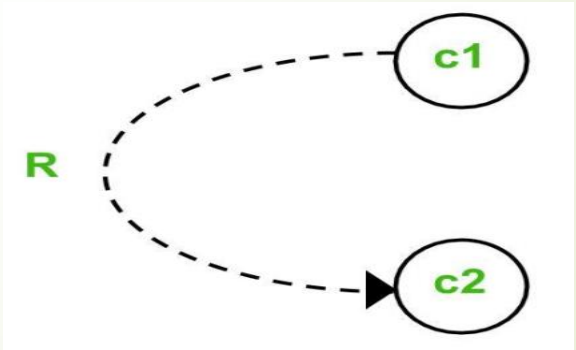


Constraints in Cause-effect graph:

- **One and Only One constraint or O-constraint:** This constraint exists between causes. It states that one and only one of c1 and c2 must be 1.

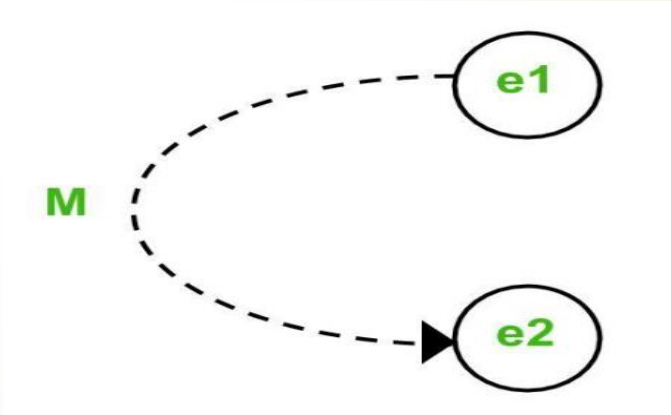


- **Requires constraint or R-constraint:** This constraint exists between causes. It states that for c1 to be 1, c2 must be 1. It is impossible for c1 to be 1 and c2 to be 0.



Constraints in Cause-effect graph:

- **Mask constraint or M-constraint:** This constraint exists between effects. It states that if effect e1 is 1, the effect e2 is forced to be 0.



Steps used in deriving test cases using this technique are:

- **Division of specification:**
  - It is difficult to work with cause-effect graphs of large specifications as they are complex, **the specifications are divided into small workable pieces** and then converted into cause-effect graphs separately.
- **Identification of cause and effects:**
  - This involves identifying the **causes(distinct input conditions)** and **effects(output conditions)** in the specification.

Steps used in deriving test cases using this technique are:

- **Transforming the specifications into a cause-effect graph:**
  - The causes and effects are linked together using Boolean expressions to obtain a cause-effect graph. Constraints are also added between causes and effects if possible.
- **Conversion into decision table:**
  - The cause-effect graph is then converted into a limited entry decision table.
- **Deriving test cases:**
  - Each column of the decision-table is converted into a test case.

Cause-effect graph Example:

- There are two columns: col1 and col2.
- The characters allowed in col1 are ‘a’ or ‘b’.
- The characters allowed in col2 are digits 0-9.
- A file is updated if both the columns i.e. col1 and col2 are correct.
- If col1 is incorrect, message ‘X’ is displayed and if col2 is incorrect, message ‘Y’ is displayed.
- Draw the cause-effect graph for the given problem.

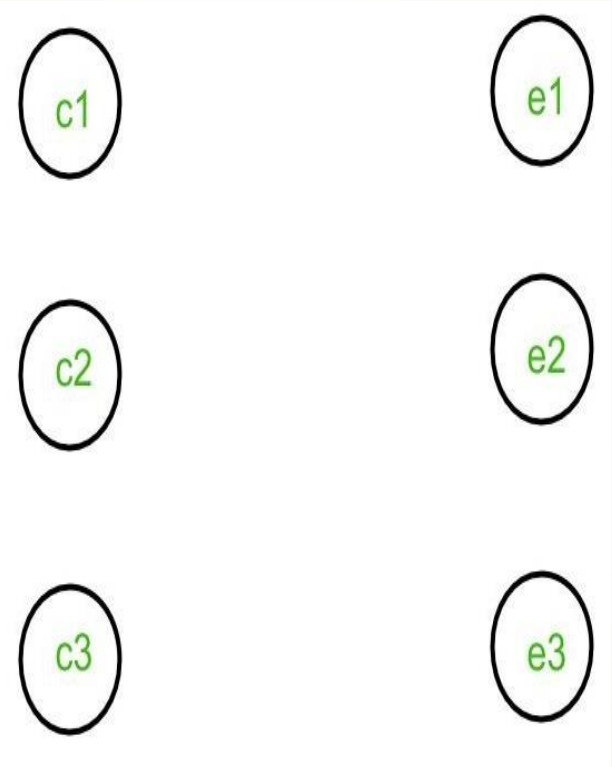
Column 1	Column 2
Correct value - A or B	*Correct value- Any digit*
Incorrect value - Any character except A or B	*Incorrect value- Any character except digit*

Cause-effect graph Example:

- **Step 1: Identify the causes:**
  - c1 - Character in column 1 is a.
  - c2 - Character in column 1 is b.
  - c3 - Character in column 2 is digit.
- **Step 2: Identify the effects:**
  - e1 - Update made (C1 OR C2) AND C3
  - e2 - Displays Message X (NOT C1 AND NOT C2)
  - e3 - Displays Message Y (NOT C3)

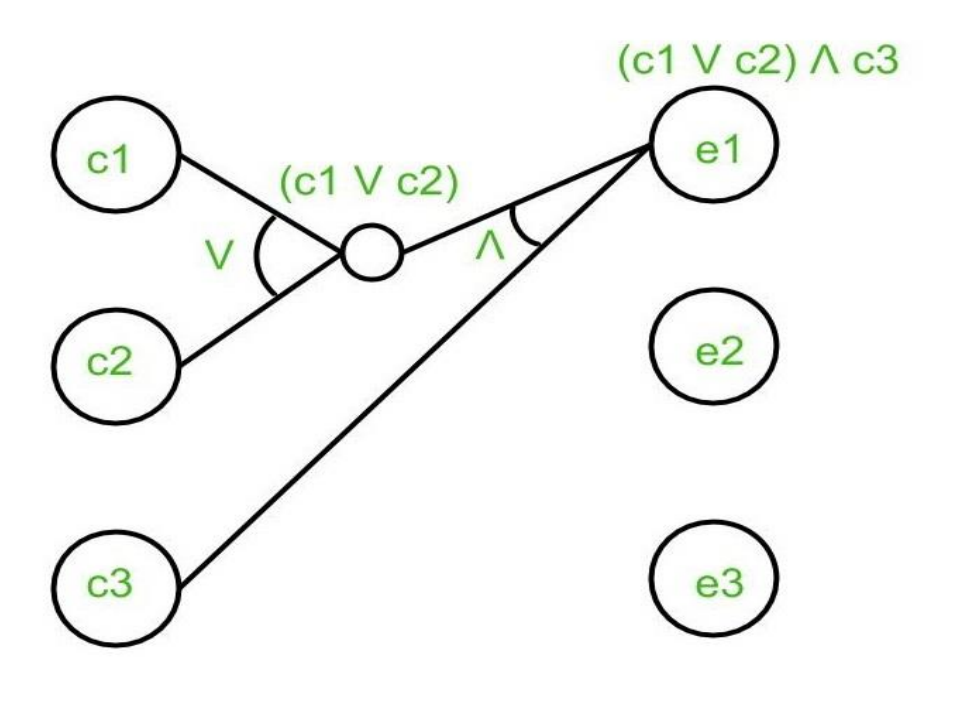
Cause-effect graph Example:

- Step 3: Create nodes for all the causes and effects:



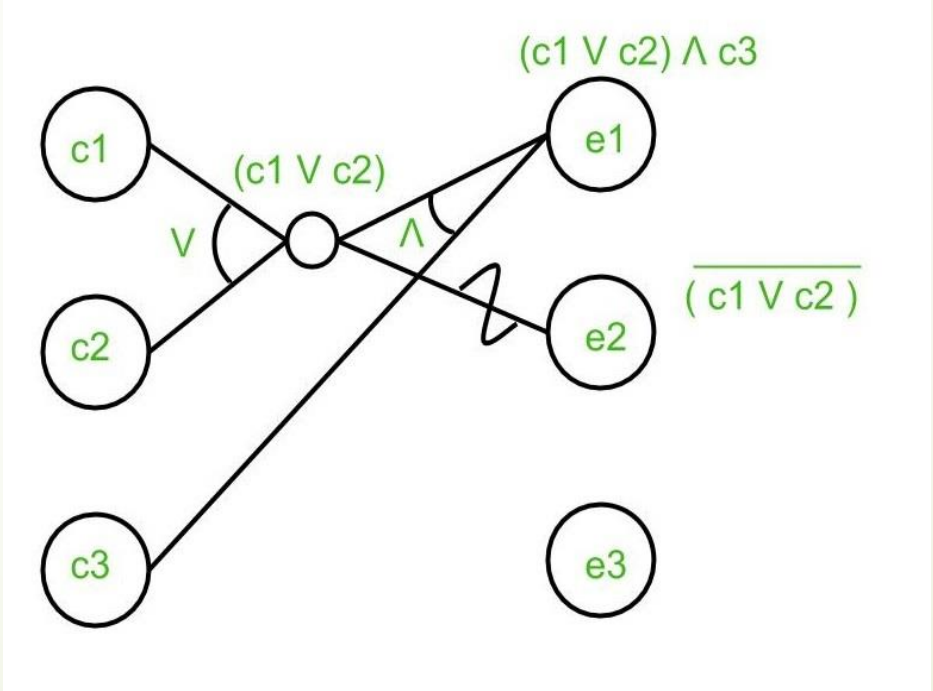
Cause-effect graph Example:

- **Step 4: Use AND, NOT, OR and Identity functions to establish links between causes and effects:**
  - e1 is obtained from c1, c2 and c3 in the following manner:



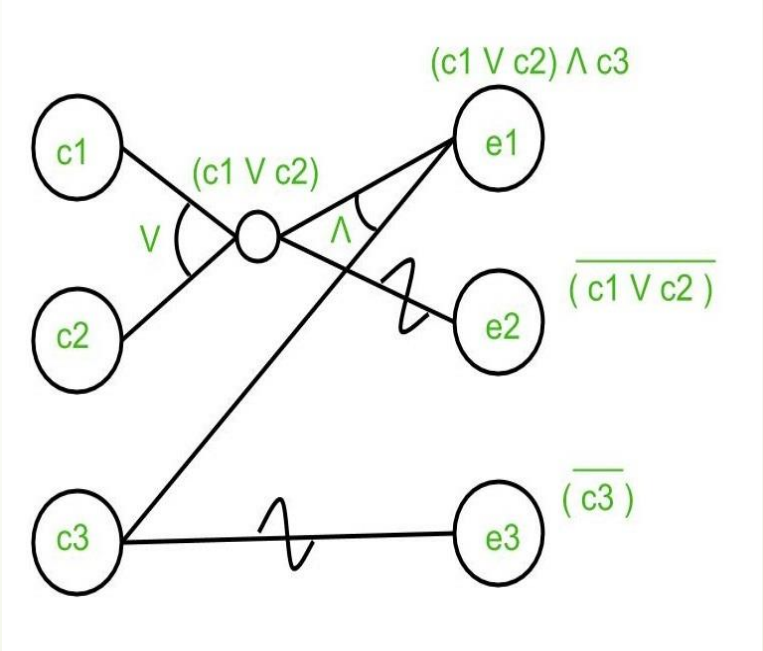
Cause-effect graph Example:

- **Step 4: Use AND, NOT, OR and Identity functions to establish links between causes and effects:**
  - e2 is obtained from c1 and c2 in the following manner:

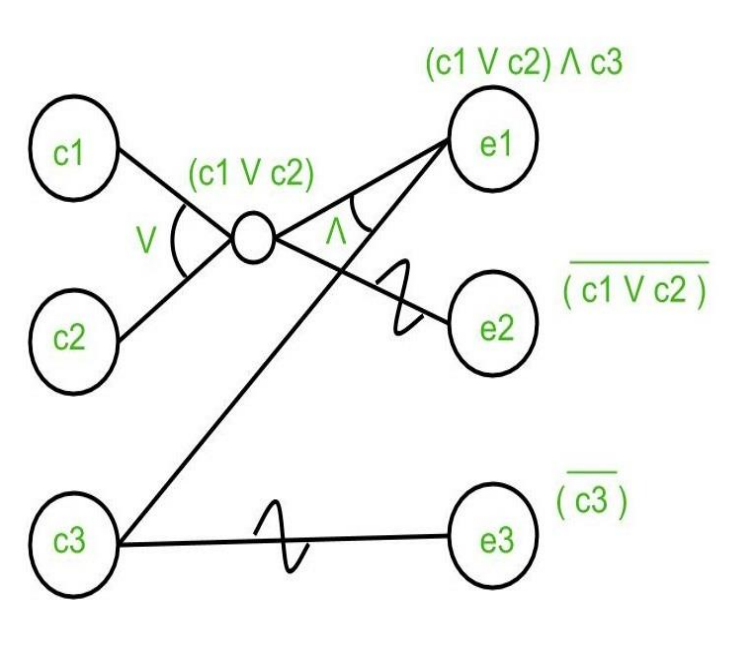


Cause-effect graph Example:

- **Step 4: Use AND, NOT, OR and Identity functions to establish links between causes and effects:**
  - e3 is obtained from c3 in the following manner:



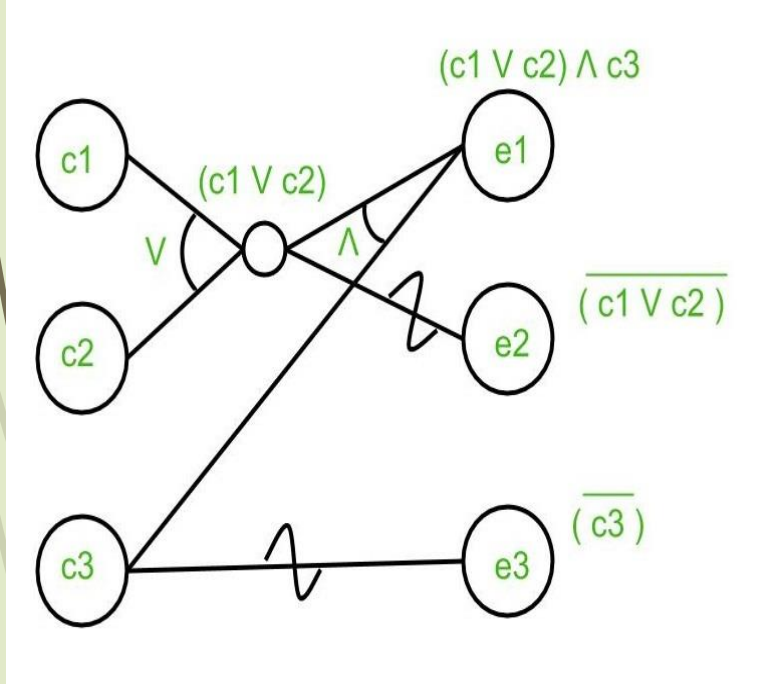
Writing Decision Table Based On Cause And Effect Graph:



Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

Actions				
C1	1		0	
C2		1		0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

Writing Decision Table Based On Cause And Effect Graph:



Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Grey Box Testing:

- **Gray box testing** is a software testing technique to test a software product or application **with partial knowledge of internal structure of the application.**
- The **purpose** of grey box testing **is to search and identify the defects due to improper code structure or improper use of applications.**
- Gray Box Testing is a software testing method, which is a **combination of both White Box Testing and Black Box Testing method.**
  - In White Box testing internal structure (code) is known
  - In Black Box testing internal structure (code) is unknown
  - In Grey Box Testing **internal structure (code) is partially known**

Grey Box Testing:

- Grey Box testers have access to the detailed design documents along with information about requirements.
- Grey Box **tests are generated based on the state-based models, UML Diagrams or architecture diagrams** of the target system.
- **Pros:**
  - Grey box testing provides combined benefits of both white box and black box testing
  - Grey-box tester can handles complex design test scenario more intelligently
  - Maintains the boundary between independent testers and developers
- Cons:
  - In grey-box testing, complete white box testing cannot be done due to inaccessible source code.

White / Black /Grey Box Testing:

- **White Box Testing:** Detailed investigation of internal logic and structure.
- **Black Box Testing:** Testing without having any knowledge of the internal working of the application.
- **Grey Box Testing:** White box + Black box = Grey box, a technique to test the application with limited knowledge of the internal working.

Static Analysis v/s Dynamic Analysis

Static Analysis

- Analyze code & generated code
- Typical use cases:
  - Syntax checking
  - Code smell detection
  - Coding standard compliance
- Typical Defects found:
  - Syntax violation
  - Unreachable code
  - Overly complicated constructs

Static Analysis v/s Dynamic Analysis

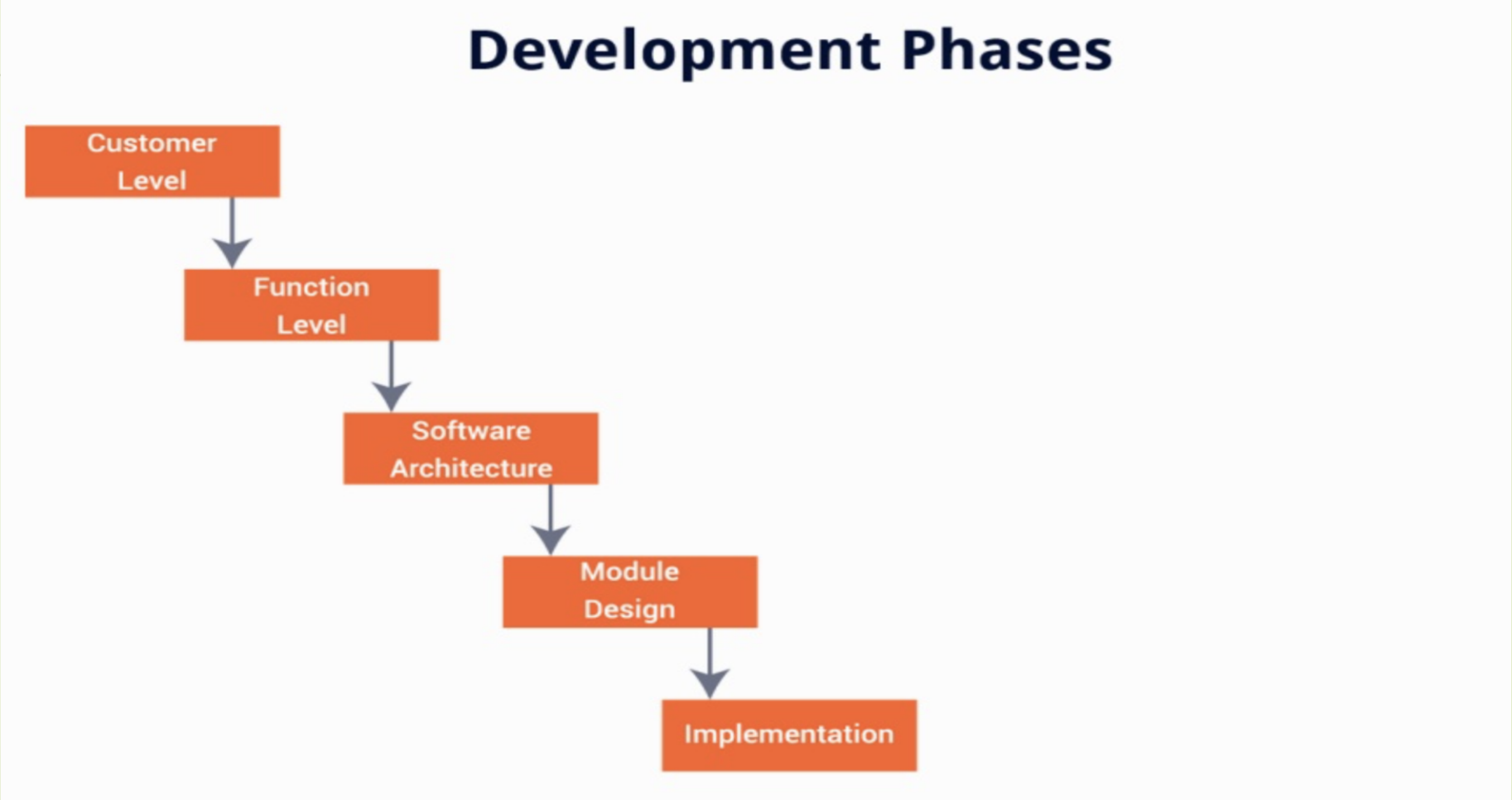
Dynamic Analysis

- Primarily Verification
- Execution of code
- Find failures
- Testing methods:
  - Black Box Testing
  - White Box Testing
- Different Test levels:

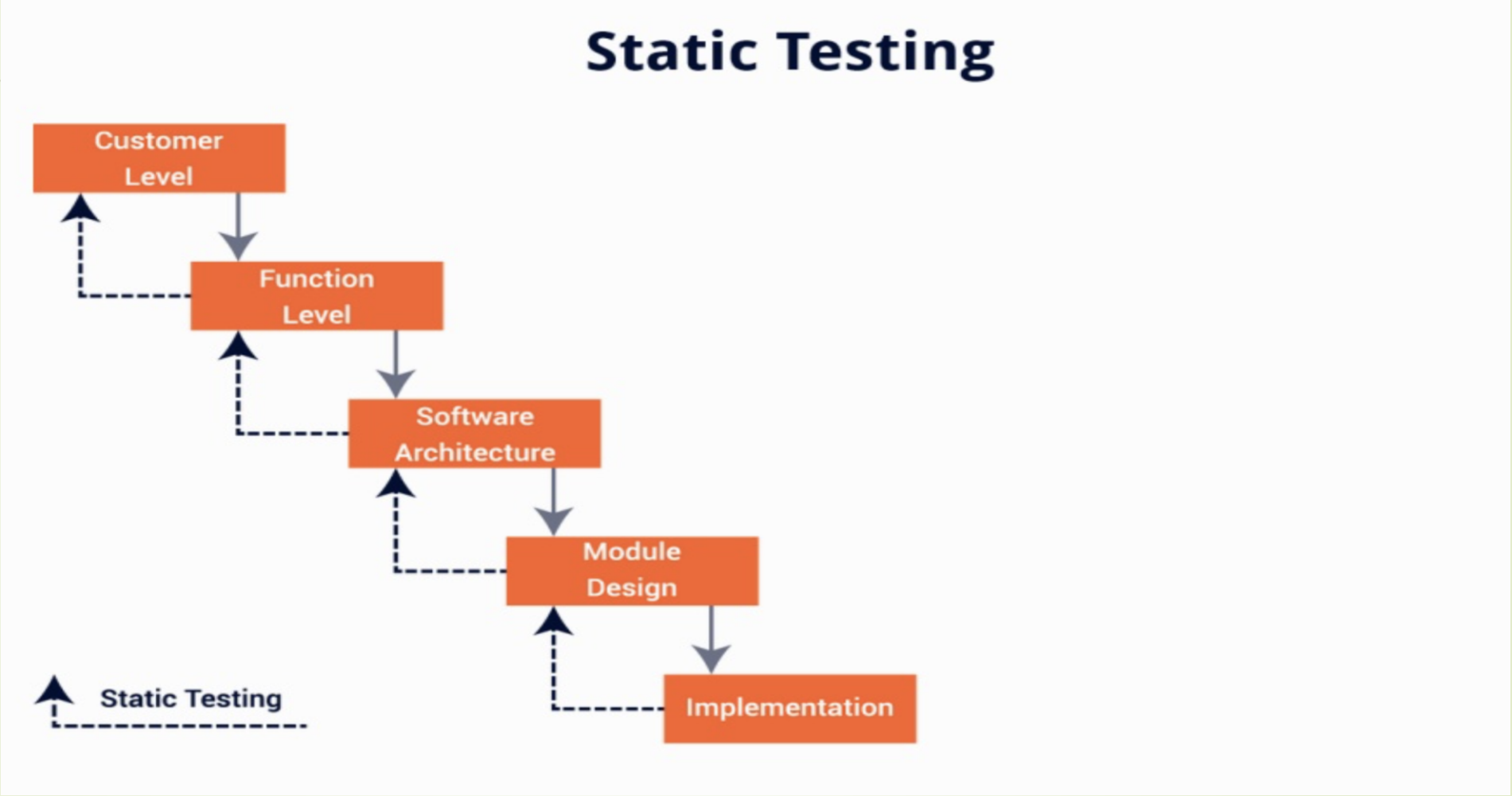
  I) Unit Testing II) Integration Testing

  III) System Testing IV) Acceptance Testing

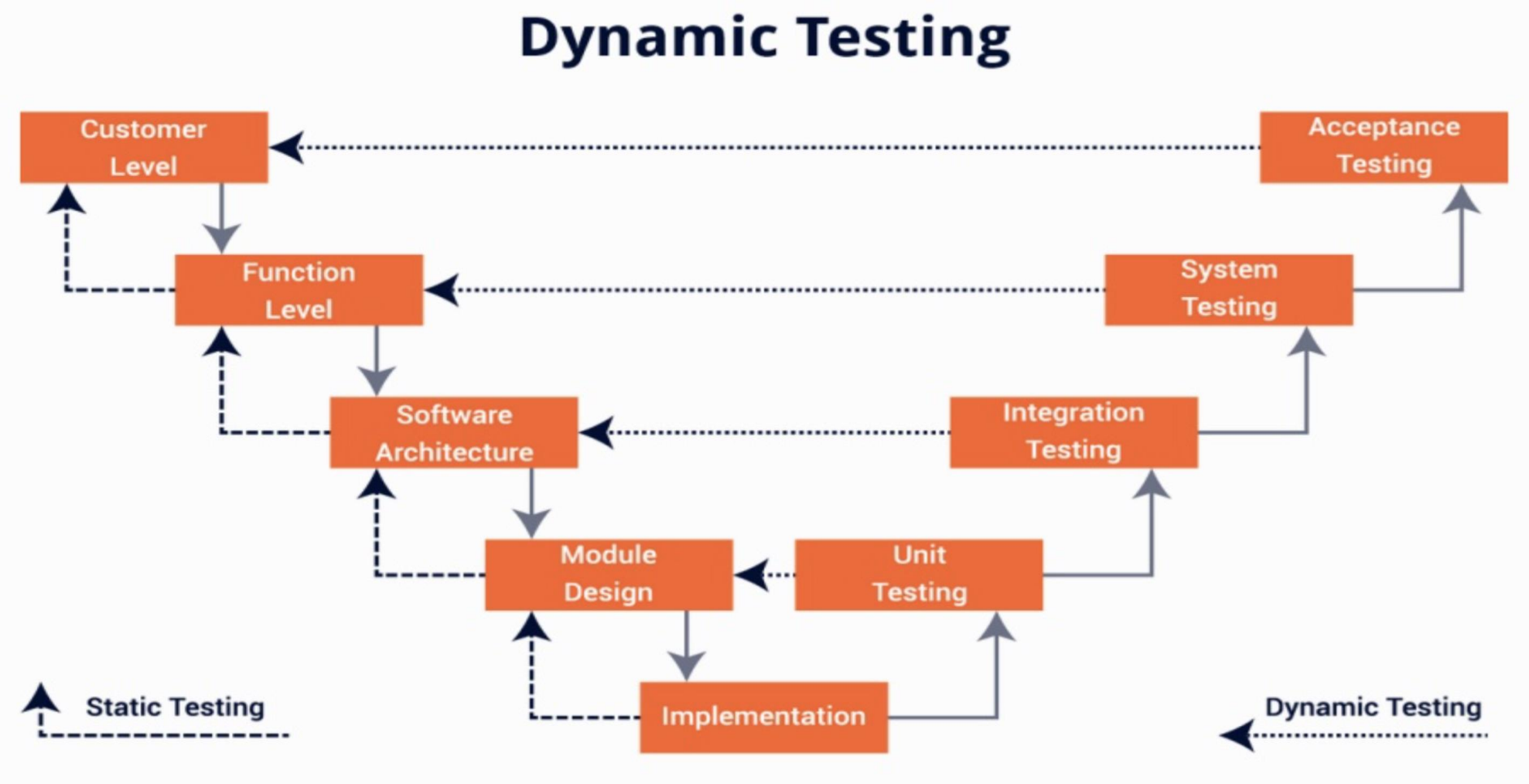
Static Analysis v/s Dynamic Analysis



Static Analysis v/s Dynamic Analysis



Static Analysis v/s Dynamic Analysis



Software Testing

- Functional Testing
- Non-functional Testing

Functional Testing

- Functional testing is a type of software testing in which the **system is tested against the functional requirements and specifications.**
- Functional testing ensures that the requirements or specifications are properly satisfied by the application. **This type of testing is particularly concerned with the result of processing.**
- It is basically defined as a type of testing which verifies that each function of the software application works in conformance with the requirement and specification.

Functional Testing

- This testing is **not concerned about the source code of the application.**
- Each functionality of the software application is tested by providing **appropriate test input, expecting the output and comparing the actual output with the expected output.**

Non-functional Testing

- Non-functional testing is a type of software testing that is performed to verify the non-functional requirements of the application.
- It verifies **whether the behavior of the system is as per the requirement or not.** It tests all the aspects which are not tested in functional testing.
- Non-functional testing is defined as a type of software testing **to check non-functional aspects of a software application.**
- It is designed **to test the readiness of a system as per nonfunctional parameters** which are never addressed by functional testing.

Categories of Testing

- **Unit testing**
- **Integration testing**
- **Validation testing**
  - Focus is on software requirements
- **System testing**
  - Focus is on system integration
- **Alpha/Beta testing**
  - Focus is on customer usage
- **Recovery testing**
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- **Security testing**
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- **Stress testing**
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- **Performance Testing**
  - test the run-time performance of software within the context of an integrated system.

Taxonomy of Software Testing:

Functional Testing

Unit Testing
Integration Testing
Regression Testing
Smoke Testing
Alpha Testing
Beta Testing
System Testing

Non-functional Testing

Stress Testing
Performance Testing
Usability Testing
Security Testing
Portability Testing

Unit Testing:

- Unit Testing is a software testing technique by means of which **individual units of software i.e. group of computer program modules, usage procedures and operating procedures are tested** to determine whether they are suitable for use or not.
- It **focuses on smallest unit** of software design.
- It is a testing method using which **every independent modules are tested** to determine if there are any issue by the developer himself. **It is correlated with functional correctness of the independent modules.**

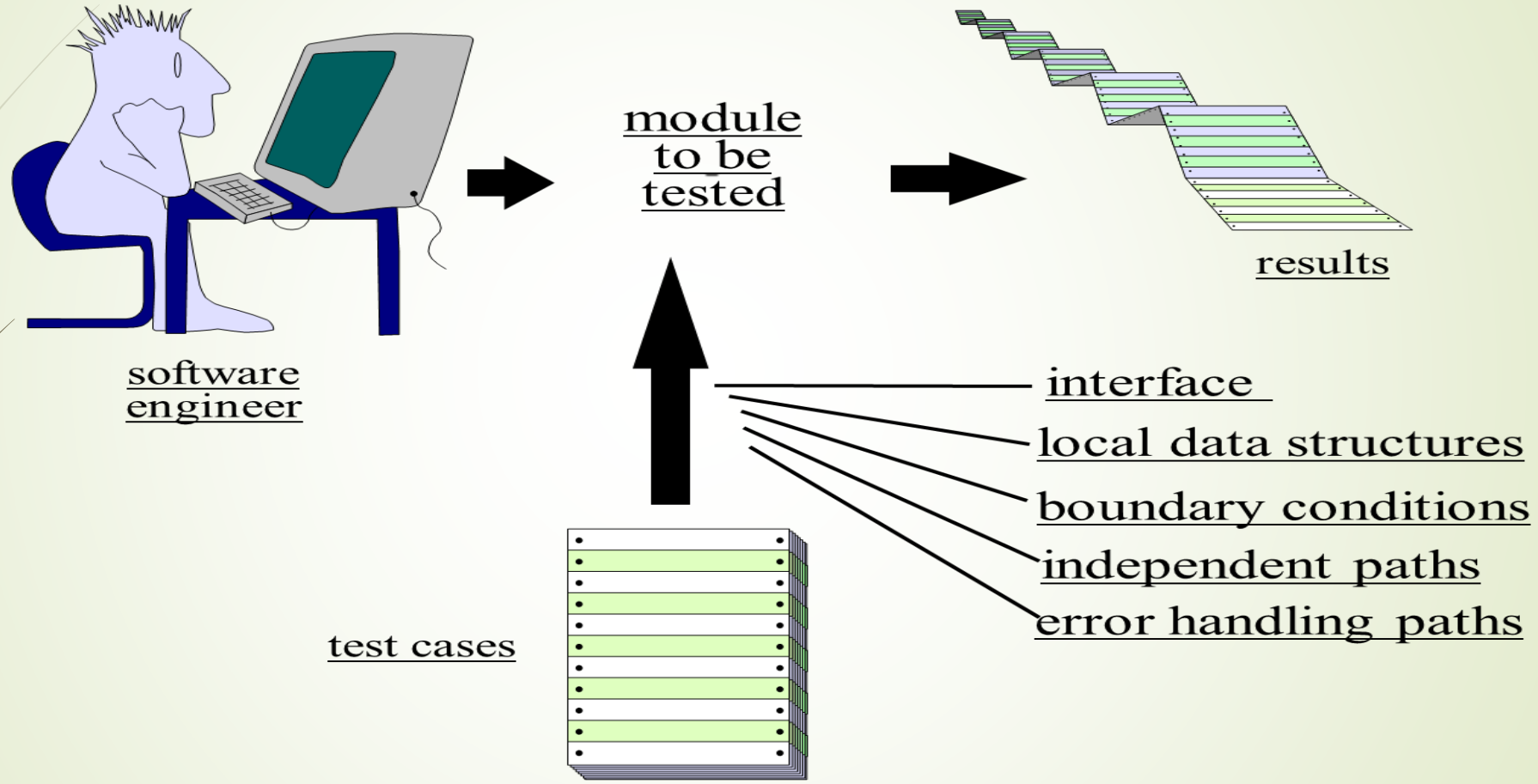
Unit Testing:

- Unit Testing is defined as a type of software testing where **individual components of a software are tested.**
- Unit Testing of software product **is carried out during the development of an application.** An individual component may be either an individual function or a procedure. **Unit Testing is typically performed by the developer.**
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Unit Testing:

- **The objective of Unit Testing is:**
  - To isolate a section of code.
  - To verify the correctness of code.
  - To test every function and procedure.
  - To fix bug early in development cycle and to save costs.
  - To help the developers to understand the code base and enable them to make changes quickly.
  - To help for code reuse.

Unit Testing:



Targets for Unit Test Cases:

- **Module interface:**
  - Ensure that information flows properly into and out of the module.
- **Local data structures:**
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution.
- **Boundary conditions:**
  - Ensure that the module operates properly at boundary values established to limit or restrict processing.
- **Independent paths (basis paths):**
  - Paths are exercised to ensure that all statements in a module have been executed at least once.
- **Error handling paths:**
  - Ensure that the algorithms respond correctly to specific error conditions.

Common Computational Errors in Execution Paths:

- Misunderstood or incorrect arithmetic precedence.
- Mixed mode operations (e.g., int, float, char).
- Incorrect initialization of values.
- Precision inaccuracy and round-off errors.
- Incorrect symbolic representation of an expression.

Other Errors to Uncover:

- Comparison of different data types
- Incorrect logical operators or precedence
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

Drivers and Stubs for Unit Testing:

Driver :

- A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results.
- A software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results.
- A test driver provides following facilities to a unit to be tested:
  - Initializes the environment desired for testing.
  - Provides simulated inputs in the required format to the units to be tested.

Drivers and Stubs for Unit Testing:

Stubs :

- Serve to replace modules that are subordinate to (called by) the component to be tested.
- It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing.

- Drivers and stubs both must be written but don't constitute part of the installed software product.
- By using stubs and driver effectively, we can cut down our total debugging and testing time by testing small parts of a program individually.

- Which of the following are facts about a top-down software testing approach**?**

I. Top down testing typically requires the tester to build method stubs.

II. Top-down testing typically requires the tester to build test drivers.

A. Only I

B. Only II

C. Both I and II

D. Neither I nor II

Integration Testing:

- **Why do we do integration testing:**
  - Unit tests only test the unit in isolation.
  - Many failures result from faults in the interaction of subsystems.
  - Often many Off-the-shelf components are used that cannot be unit tested.
  - Without integration testing the system test will be very time consuming.
  - Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

Integration Testing:

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces.
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing.
    - Big-Bang approach
  - Incremental Integration Testing.
    - Top down and Bottom approach

Non-incremental Integration Testing

- Commonly called the **"Big Bang"** approach
- All components are combined in advance
- The entire program is tested as a whole
- Chaos (Confuse or unpredictable) results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop.

Incremental Integration Testing:

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments.
- Errors are easier to isolate and correct.
- Interfaces are more likely to be tested completely.
- A systematic test approach is applied.

Top-down Integration:

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module.
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion.
- After testing the top module, stubs are replaced one at a time with the actual modules for integration.
- Perform testing on this recent integrated environment.
- Regression testing may be conducted to ensure that new error have not appeared.

Top-down Integration:

- **Disadvantages**
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded.
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process.

Bottom-up Integration:

- Integration and testing starts with the lowest level modules in the control hierarchy. These are the modules from which no other module is being called.
- Design the driver module for the superordinate module which calls the module selected in step 1.
- Test the module selected in step 1 with the driver designed in step 2.
- The next module to be tested is any module whose subordinate modules have all been tested.

Bottom-up Integration:

- **Disadvantages**
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available.

Sandwich Integration:

- Consists of a combination of both top-down and bottom-up integration.
- Occurs both at the highest level modules and also at the lowest level modules.
- Proceeds using functional groups of modules, with each group completed before the next
- High and low-level modules are grouped based on the control and data processing they provide for a specific program feature

Sandwich Integration:

- Integration within the group progresses in alternating steps between the high and low level modules of the group
- When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs.
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario.

Regression Testing:

- Each new addition or change to base lined software may cause problems with functions that previously worked flawlessly.
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools

Regression Testing:

- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

Smoke Testing:

- Smoke Testing is a software testing process that **determines whether the deployed software build is stable or not.**
- Smoke testing is a confirmation for QA team to proceed with further software testing.
- It consists of a **minimal set of tests run on each build to test software functionalities.**
- The QA team test the application against the critical functionalities. These series of test cases are designed to expose errors that are in build. If these tests are passed, QA team continues with another testing.
- Smoke Testing is also known as **Confidence Testing** or **Build Verification Testing.**

Alpha Testing:

- It is a type of acceptance testing which is done **before the product is released to customers.** It is typically done by QA people.
- The **main objective** of alpha testing is **to refine the software product by finding and fixing the bugs** that were not discovered through previous tests
- It is the final testing stage before the software is released into the real world.

Alpha Testing:

- Alpha testing has two phases,
- The **first phase** of testing is done by **in-house developers.** They either use hardware-assisted debuggers or debugger software.
- The aim to catch bugs quickly. Usually while alpha testing, a tester will come across a plenty of bugs, crashes, missing features, and docs.
- While the **second phase** of alpha testing is done by **software QA staff**, for additional testing in an environment.

Beta Testing:

- The beta test is **conducted at one or more customer sites by the end-user of the software.** This version is released for the limited number of users for testing in real time environment.
- Beta version of the software, whose feedback is needed, **is released to a limited number of end-users of the product to obtain feedback on the product quality.**
- Beta testing helps in minimization of product failure risks and it provides increased quality of the product through **customer validation.**

System Testing:

- System Testing is basically **performed by a testing team that is independent of the development team** that helps to test the quality of the system impartial. **It has both functional and non-functional testing.**
- In this **software is tested such that it works fine for different operating system.**
- It is covered under the **black box testing** technique. In this we just focus on required input and output without focusing on internal working.

Non-Functional Testing

Stress Testing:

- Stress Testing includes **testing the behavior of a software under abnormal conditions.** For example, it may include taking away some resources or applying a load beyond the actual load limit.
- It is a software testing technique that **determines the robustness of software** by testing beyond the limits of normal operation.
- Stress testing is particularly **important for critical software** but is used for all types of software.

Stress Testing:

- Stress testing **emphasizes on robustness, availability and error handling under a heavy load** rather than on what is correct behavior under normal situations.
- **The aim of stress testing is** to test the software by applying the load to the system and taking over the resources used by the software **to identify the breaking point.**

Performance Testing:

- It is designed to test the run-time performance of software within the context of an integrated system. **It is used to test speed and effectiveness of program.**

- It is also called **load testing.** In it we check, **what is the performance of the system in the given load.**

Usability Testing:

- Usability Testing also known as **User Experience(UX) Testing**, is a testing method **for measuring how easy and user-friendly a software application is.**
- A small set of target end-users, use software application **to expose usability defects.**
- Usability testing mainly focuses on user's ease of using application, flexibility of application to handle controls and ability of application to meet its objectives.

Usability Testing:

- There are many software applications/websites, which miserably fail, once launched, due to following reasons –
  - Where do I click next**?**
  - Which page needs to be navigated**?**
  - Which Icon represents what**?**
  - Error messages are not consistent or effectively displayed
  - Session time not sufficient.

Security Testing:

- Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view.
- It should ensure:
  - Confidentiality
  - Integrity
  - Authentication
  - Availability
  - Authorization
  - Non-repudiation

Portability Testing:

- Portability testing includes **testing a software with the aim to ensure its reusability** and that it can be moved from another software as well.
- This includes **testing of a software w.r.t its usage over different environments** like Computer hardware, operating systems, and browser.

Thank You.