Unit 4
Software Engineering

Prepared By

Abhishek Kesharwani

Assistant Professor ,United College of Engineering and Research

SOFTWARE TESTING

What is Testing?

Many people understand many definitions of testing-

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect.

What is Testing?

A more appropriate definition is-

" Testing is the process of executing a program with the intent of finding errors ".

Why should we test?

- Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in system where human safety is involved.

- In the software life cycle, the earlier the errors are discovered and removed, the lower is the cost of their removal.

Who should do the testing?

- The testing requires the developers to find errors from their software.
- It is very difficult for software developer to point out errors from own creations.
- Many organizations have made a distinction between development and testing phase by making different people responsible for each phase.

What should we test?

- We should test the program's responses to every possible input.
- It means, we should test for all valid and invalid inputs.

Testing Objectives

- Software Testing has different objectives. The major objectives of Software testing are as follows:

1. Finding defects which may get created by the programmer while developing the software.
2. To make sure that the **end result meets the user requirements.**
3. To ensure that it **satisfies the SRS** that is System Requirement Specifications.
4. To **gain the confidence of the customers** by providing them a quality product.

Principles of Software Testing

1. Testing shows presence of defects.
2. Exhaustive testing is impossible
3. Early testing
4. Defect clustering
5. The Pesticide Paradox
6. Testing is context dependent
7. Absence of errors fallacy.

1. TESTING SHOWS THE PRESENCE OF BUGS

Testing an application can only reveal that one or more defects exist in the application, however, testing alone cannot prove that the application is error free.

2. EXHAUSTIVE TESTING IS IMPOSSIBLE

Unless the application under test has a very simple logical structure and limited input, it is not possible to test all possible combinations of data and scenarios. For this reason, risk and priorities are used to concentrate on the most important aspects to test.

3. EARLY TESTING

The sooner we start the testing activities the better we can utilize the available time. As soon as the initial products, such the requirement or design documents are available, we can start testing

4. DEFECT CLUSTERING

During testing, it can be observed that most of the reported defects are related to small number of modules within a system. i.e. small number of modules contain most of the defects in the system. This is the application of the Pareto Principle to software testing: approximately 80% of the problems are found in 20% of the modules.

5. THE PESTICIDE PARADOX

If you keep running the same set of tests over and over again, chances are no more new defects will be discovered by those test cases. Because as the system evolves, many of the previously reported defects will have been fixed and the old test cases do not apply anymore. Anytime a fault is fixed or a new functionality added, we need to do regression testing to make sure the new changed software has not broken any other part of the software. However, those regression test cases also need to change to reflect the changes made in the software to be applicable and hopefully fine new defects.

6. TESTING IS CONTEXT DEPENDENT

Different methodologies, techniques and types of testing is related to the type and nature of the application. For example, a software application in a medical device needs more testing than a games software. More importantly a medical device software requires risk based testing, be compliant with medical industry regulators and possibly specific test design techniques. By the same token, a very popular website, needs to go through rigorous performance testing as well as functionality testing to make sure the performance is not affected by the load on the servers.

7. ABSENCE OF ERRORS FALLACY

Just because testing didn't find any defects in the software, it doesn't mean that the software is ready to be shipped. Were the executed tests really designed to catch the most defects? or where they designed to see if the software matched the user's requirements? There are many other factors to be considered before making a decision to ship the software.

Software Testing Process

The most widely used testing process(levels) consists of three stages that are as follows:

1. Unit Testing
2. Integration Testing
3. System Testing

Unit Testing

- **Unit testing** is a method by which individual units of source code are tested to determine if they are fit for use.
- A unit is the smallest testable part of an application like functions, classes, procedures, interfaces.
- The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly.

Advantages of Unit Testing

1. Issues are found at early stage.
2. Unit testing helps in maintaining and changing the code.
3. Since the bugs are found early in unit testing hence it also helps in reducing the cost of bug fixes.

Component Testing

- It is also called as **module testing**.
- The basic difference between the unit testing and component testing is in unit testing the developers test their piece of code but in component testing the whole component is tested.

Stubs

Stub – the dummy modules that simulates the low level modules.

Stubs are always distinguish as "**called programs**“.

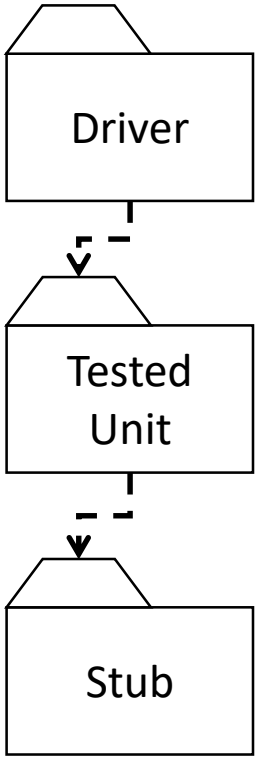
Test stubs are programs that simulate the behaviors of software components that a module undergoing tests depends on.

Drivers

- **Driver** – the dummy modules that simulate the high level modules.
- **Drivers** are also considered as the form of dummy modules which are always distinguished as "**calling programs**", that is handled in bottom up integration testing, it is only used when main programs are under construction.

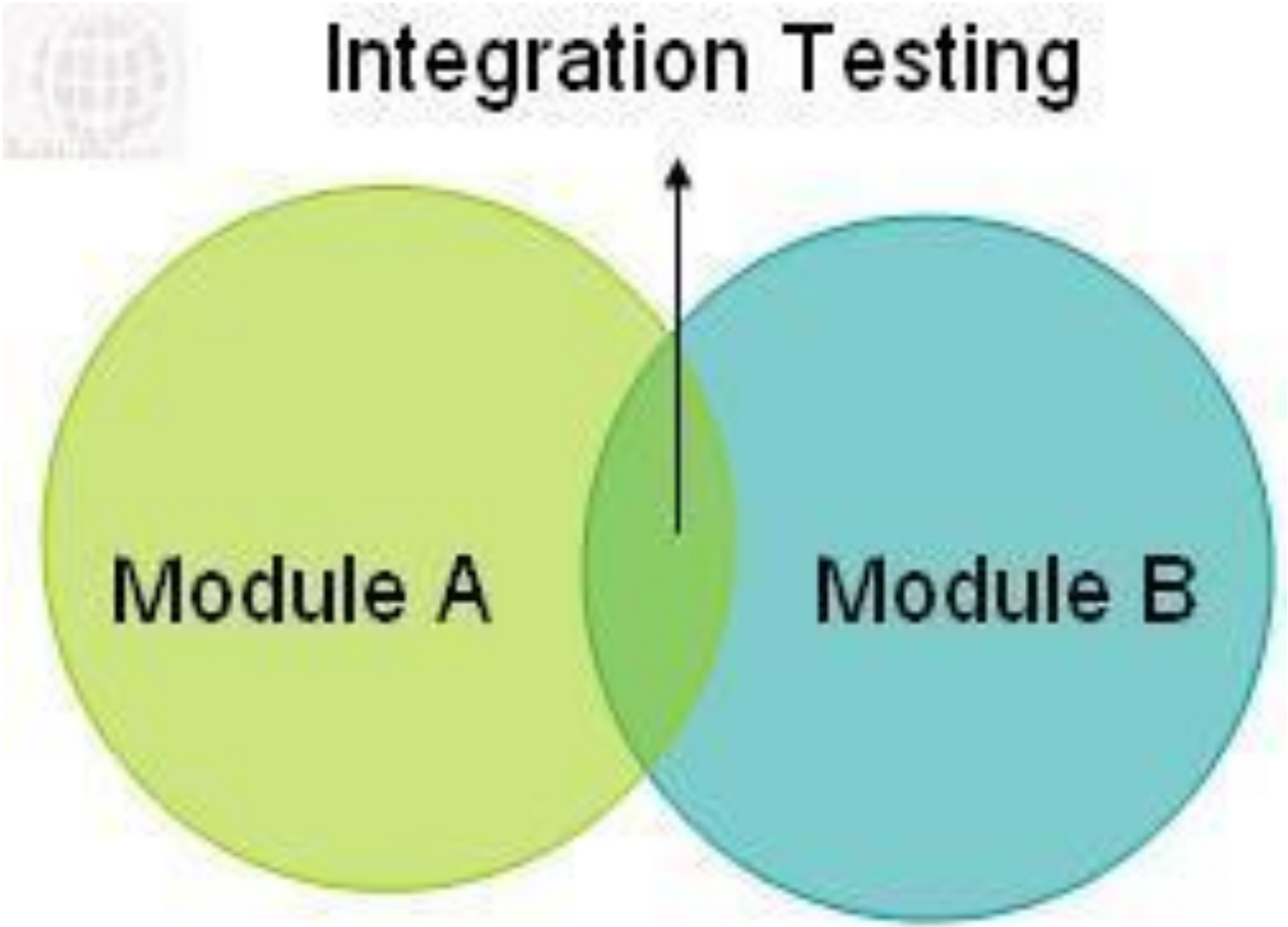
Stubs and drivers

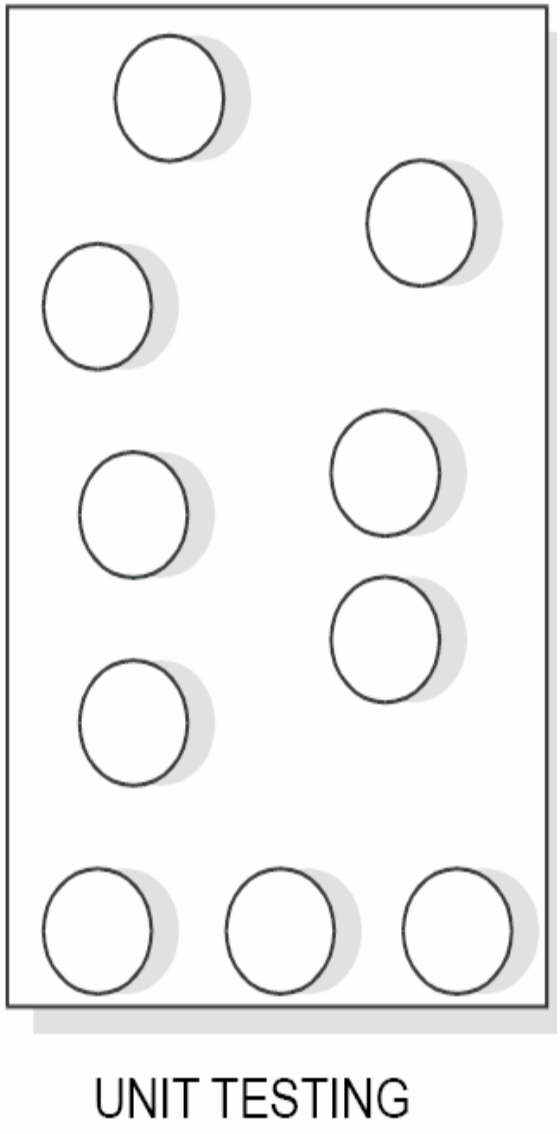
- Driver:
  - A driver calls the component to be tested
  - A component, that calls the `TestedUnit`
- Stub:
  - A stub is called from the software component to be tested
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.



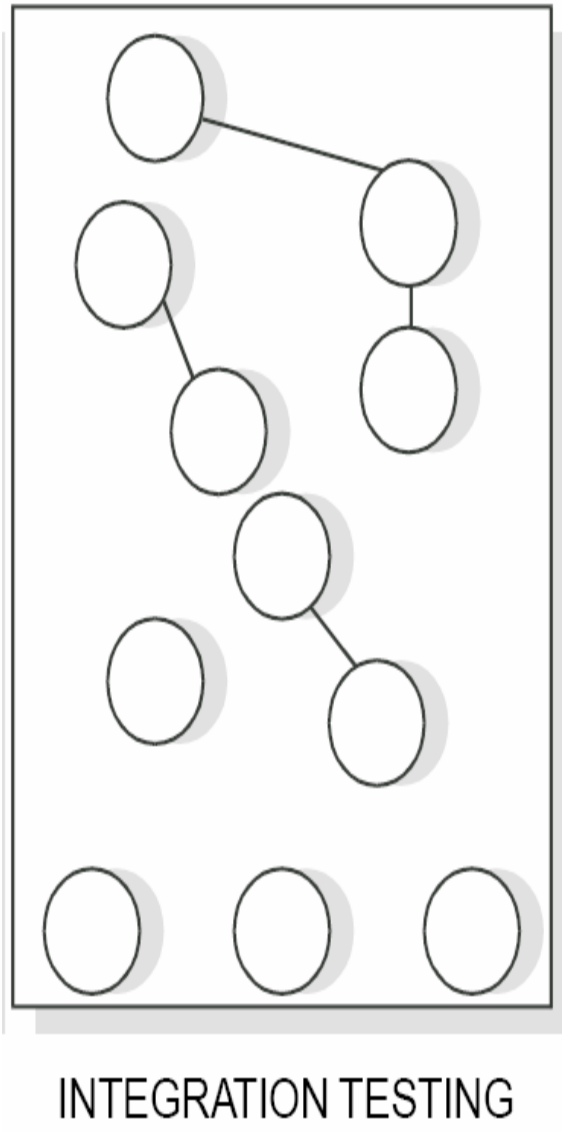
What Is Integration Testing?

- Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.
- It occurs **after** unit testing and **before** system testing.
- Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

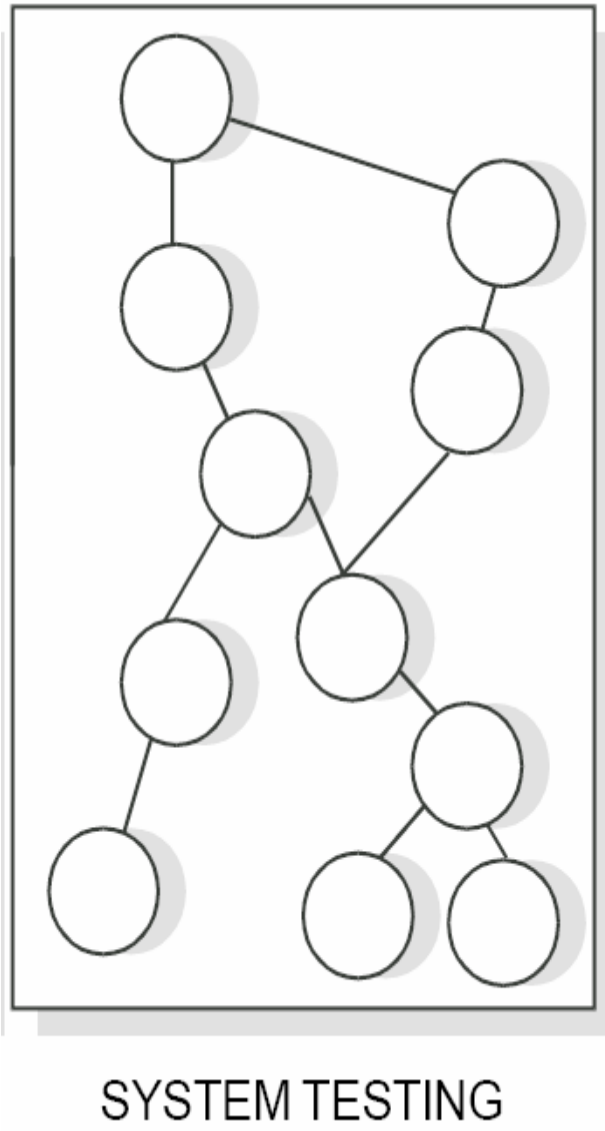

Integration Testing
Module A
Module B



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

Integration Testing Strategy

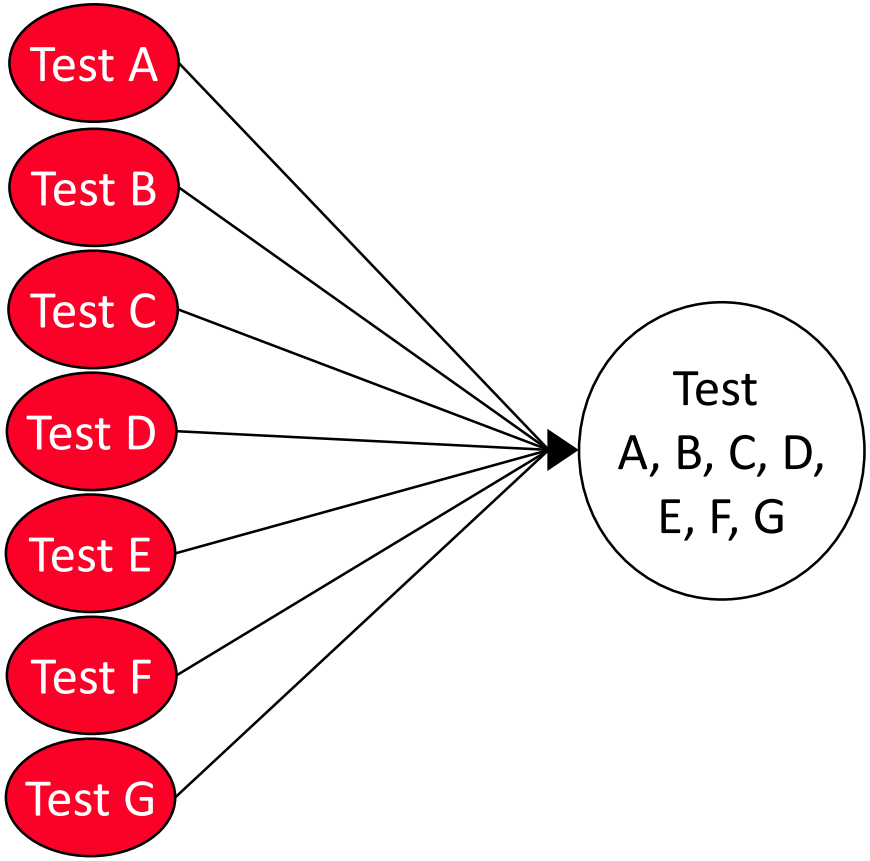
The Integration testing strategy determines the order in which the subsystems are selected for testing and integration-

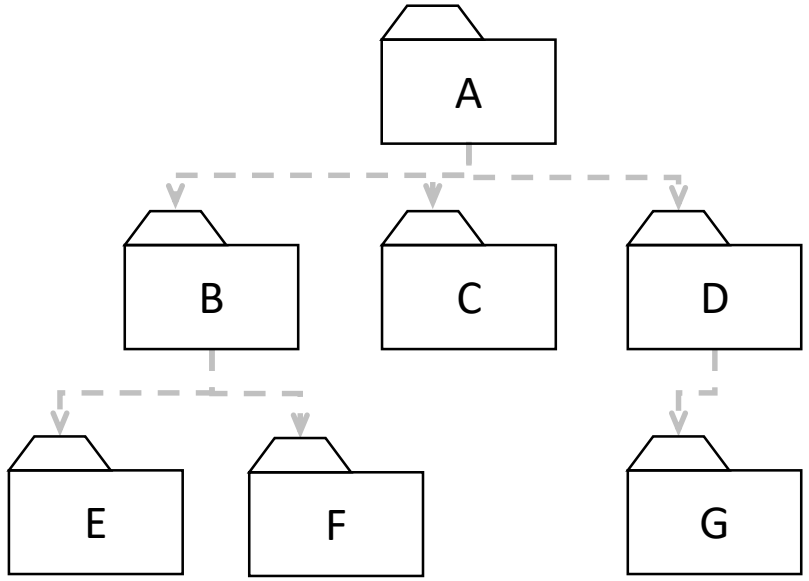
- Big bang integration (Non incremental)
- Incremental integration
- Top down integration
- Bottom up integration
- Sandwich testing

Big Bang Integration

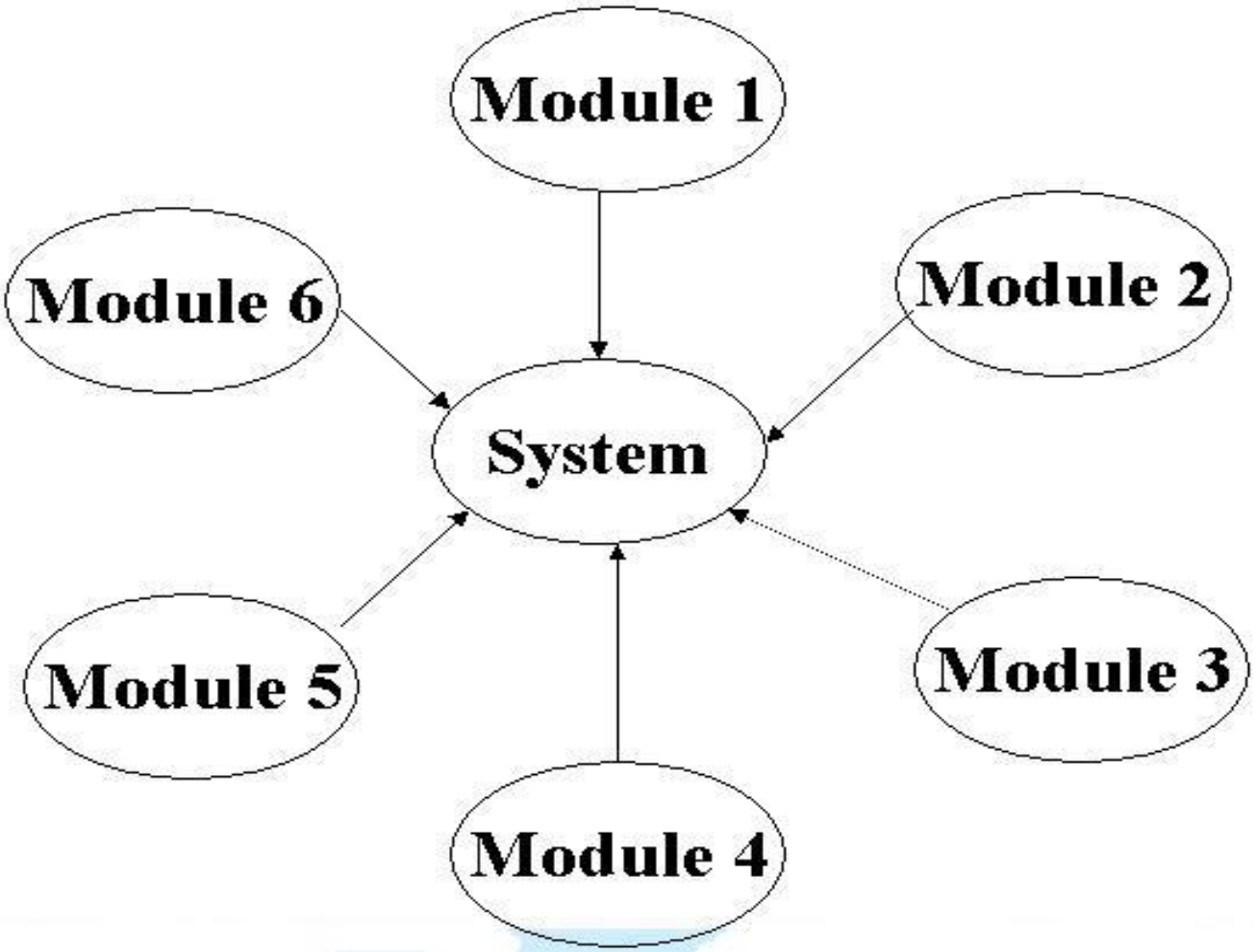
- All the components of the system are integrated & tested as a single unit.
- Instead of integrating component by component and testing, this approach waits till all components arrive and one round of integration testing is done.
- **It reduces testing effort, and removes duplication in testing.**

Big-Bang Approach



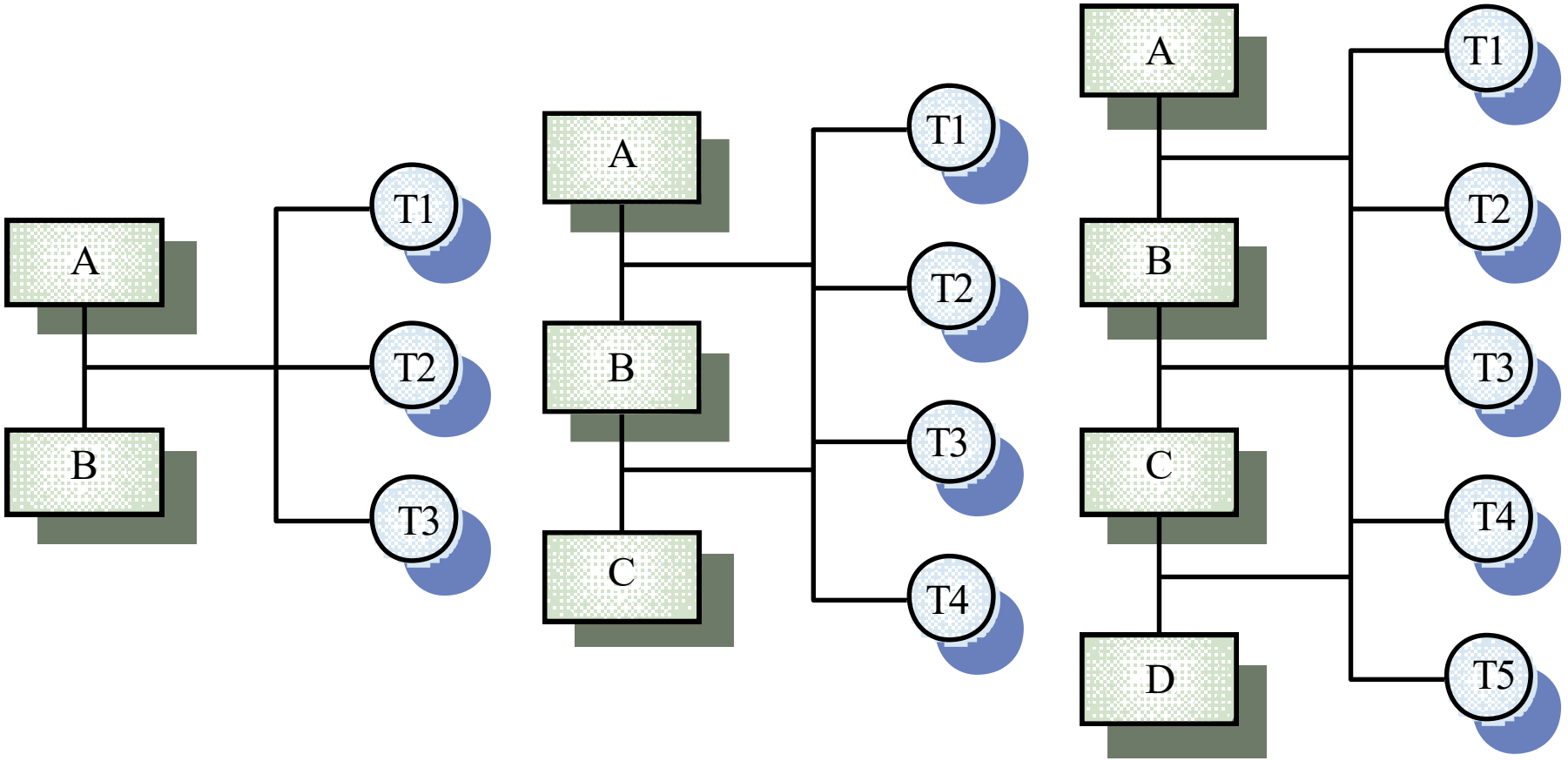


Big Bang Integration Testing



Incremental Integration

- The incremental approach means to first combine only two components together and test them. Remove the errors if they are there, otherwise combine another component to it and then test again, and so on until the whole system is developed.
- In this, the program is constructed and tested in small increments, where errors are easier to isolate and correct.



Test sequence 1

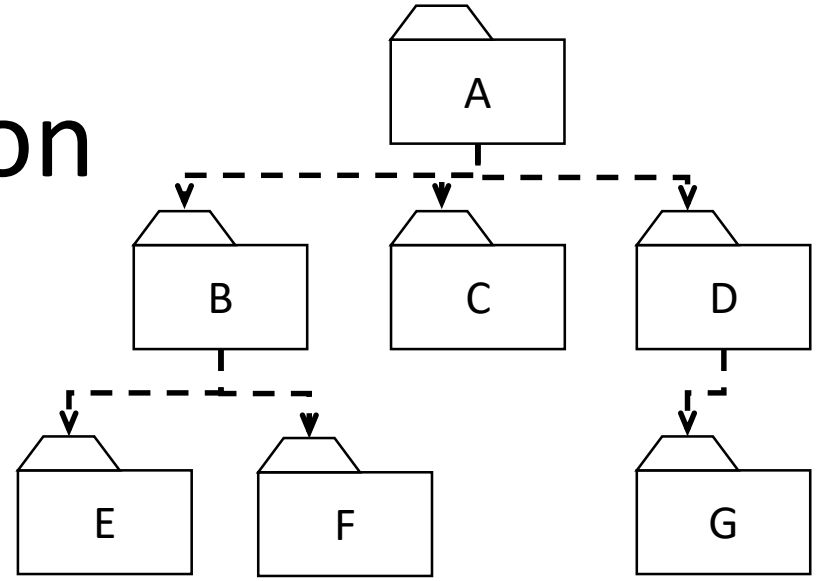
Test sequence 2

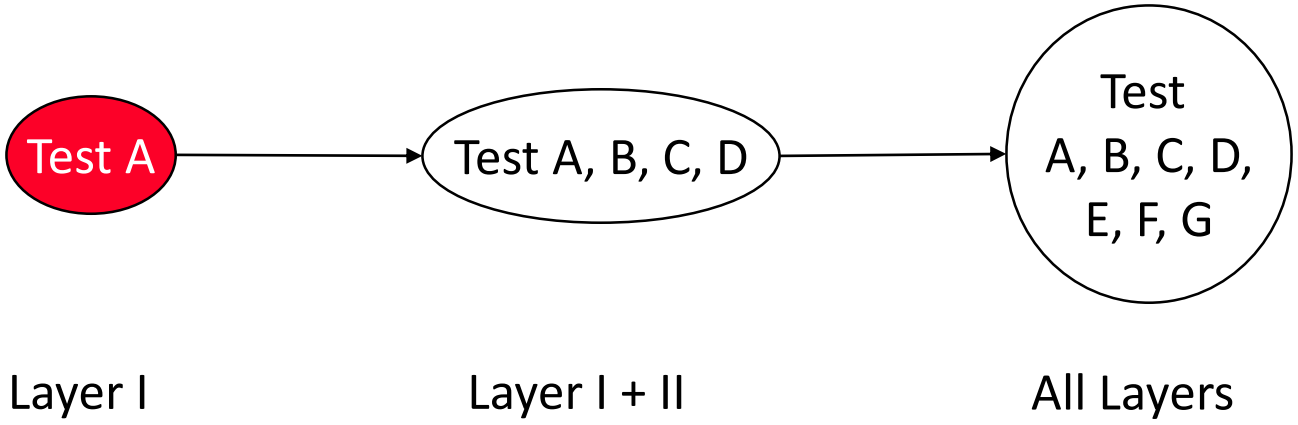
Test sequence 3

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.

Top-down Integration





Top-down Integration

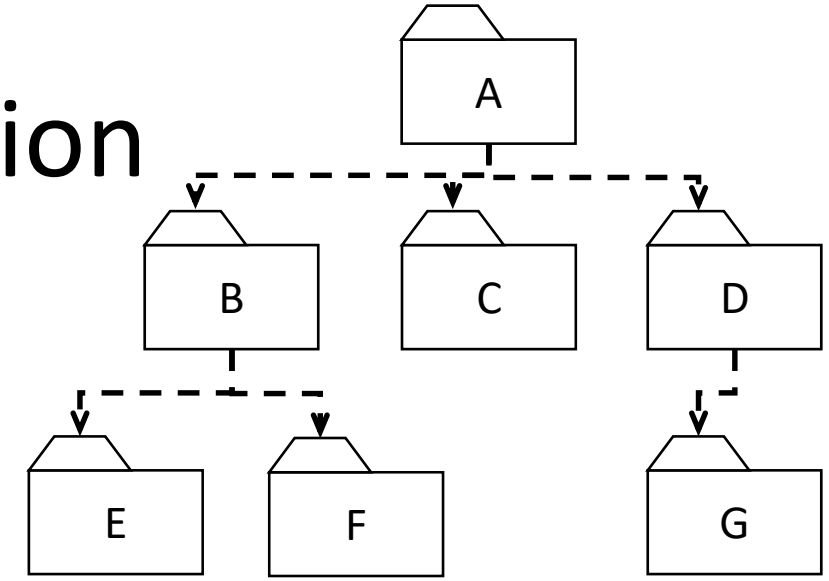
-**Breadth First Integration**(B-C-D, E-F-G): This would integrate all components on a major control path of the structure.

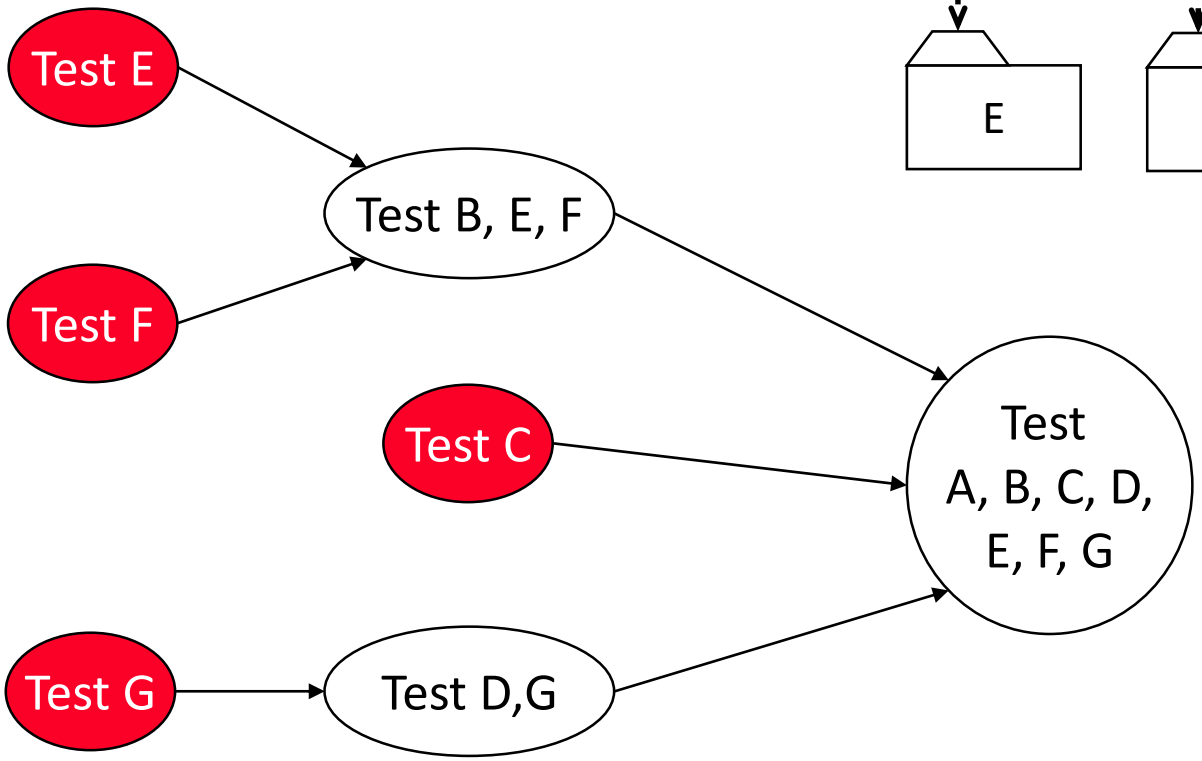
-**Depth First Integration** (A-B-E,A-B-F): This incorporates all components directly subordinate at each level, moving across the structure horizontally.

Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is repeated until all subsystems are included
- Drivers are needed.
- As integration moves upward, the need for separate test drivers lesser.

Bottom-up Integration

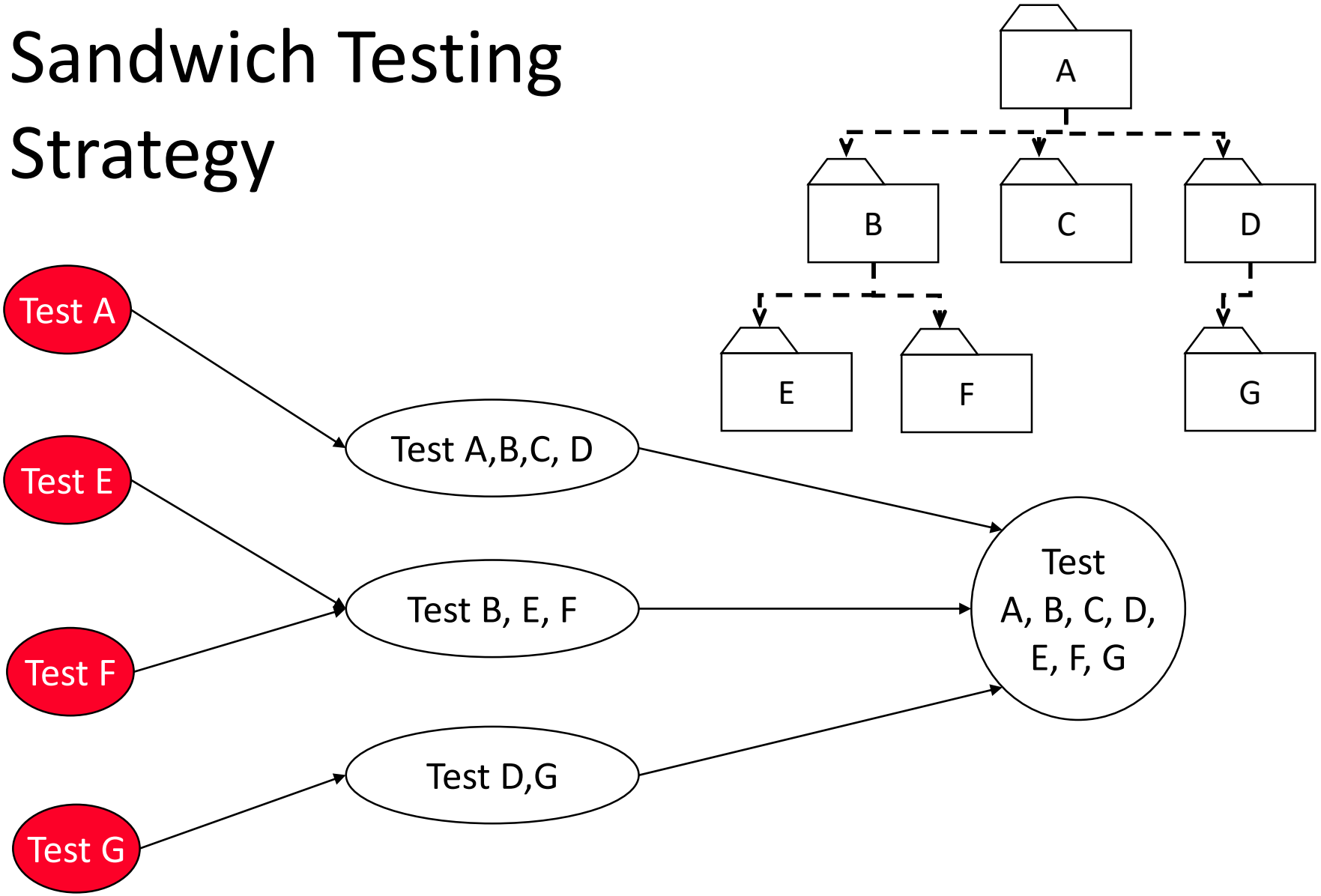


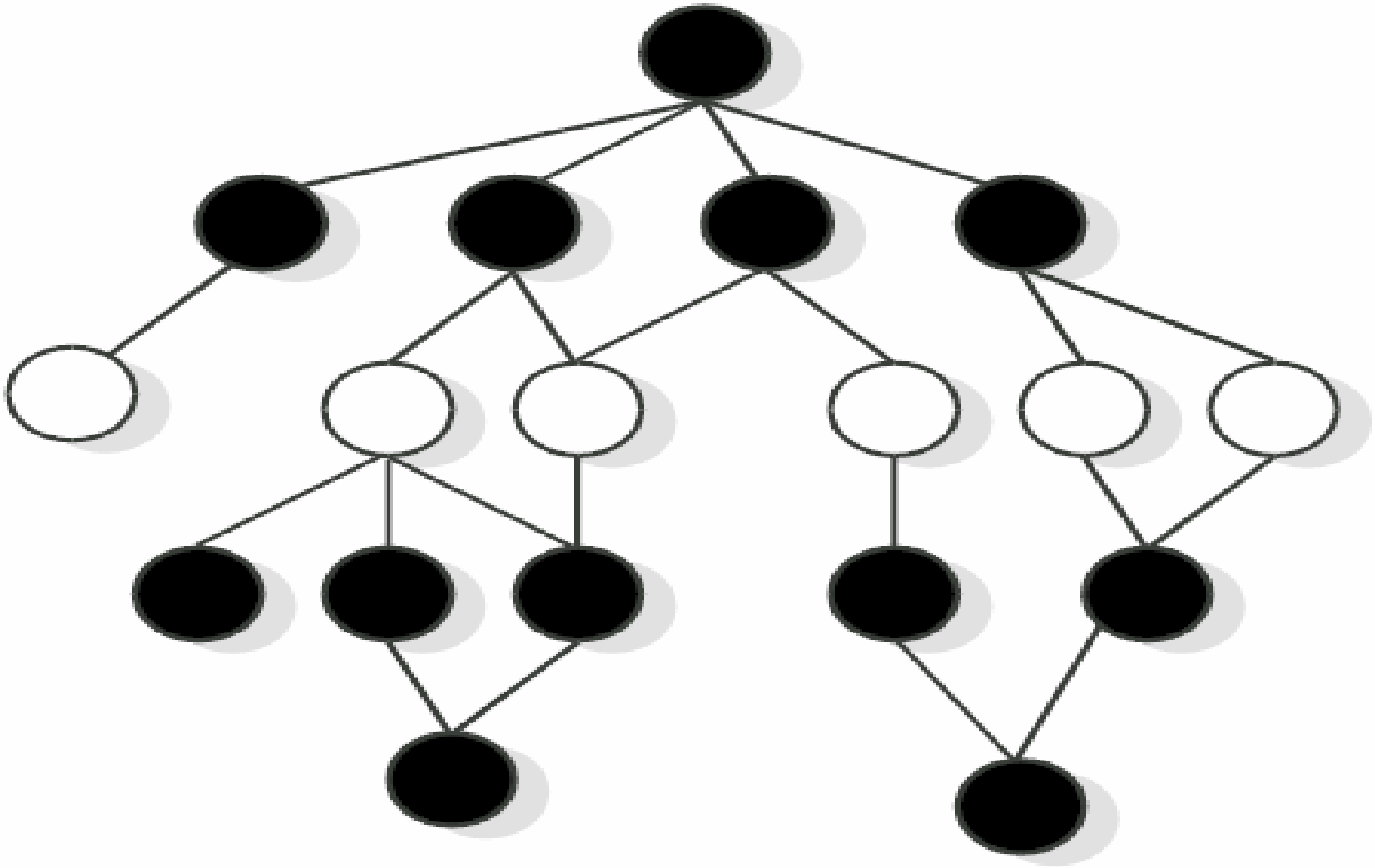


Sandwich/ Bidirectional Testing

- Combines top-down strategy with bottom-up strategy
- The system is viewed as having three layers
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
- Testing converges at the target layer.

Sandwich Testing Strategy





Sandwich integration

System Testing

- System testing is most often the final test to verify that the system to be delivered meets the specification and its purpose.
- It **tests both functional & non-functional aspects of the product.**
- It is started once unit testing , component testing & integration testing are completed.
- System testing is carried out by specialists testers or independent testers

Types of System Testing

There are essentially three main kinds of system testing –

1. Alpha Testing
2. Beta Testing
3. Acceptance Testing

Alpha Testing

- This **test takes place at the developer's site.**
- Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.

It has two phases:

- The software is tested by in-house developers.
- The software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

Beta Testing

- It takes place at **customer's site**. It is the system testing **performed by a selected group of friendly customers**.
- It sends the system to users who install it and use the software in testing mode., that is not live usage.
- The **goal of beta testing** is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final, released version of the application.

Differences between Alpha and Beta Testing

Alpha Testing	Beta Testing
It is done at developer's site.	It is done at one or more customer's site.
It is conducted in a controlled environment with developer.	It is conducted in an environment that cannot be controlled by the developer.
During alpha testing, developers records errors and usage problems.	During beta testing, customer records all problems and submits report to the developer for modifications at regular intervals.

Acceptance Testing

- It is a type of testing carried out in order to verify if the product is developed as per the standards and specifies criteria and meets all the requirements specified by customer.
- To determine whether to accept or reject the delivery of the system.
- The **goal** of acceptance testing is **to establish confidence in the system.**

Regression Testing

- It is a type of testing carried out to ensure that changes made in the fixes are not impacting the previously working functionality.
- The main aim of regression testing is to make sure that changed component is not impacting the unchanged part of the component.
- It means **re-testing** an application after its code has been modified to verify that it still functions correctly.

Types of Testing Strategy

1. Top down Strategy

   It is an approach where modules are developed and tested starting at the top level of the programming hierarchy and continuing with the lower levels.

2. Bottom up Strategy

   It is opposite of top down method. This process starts with building and testing the low level modules first, working its way up the hierarchy.

White Box testing

- White Box testing is based on the inner workings of an application and revolves around internal testing.
- The term "whitebox" was used because of the see-through box concept.
- One of the **basic goal** of white box testing is to verify a working flow for an application.

White-box testing is also called as:

- -Structural testing
- -Code Based Testing
- -Clear testing
- -Open testing
- - Glass box testing

How do you perform White Box Testing?

Testers divided it into **two basic steps:**

STEP 1) UNDERSTAND THE SOURCE CODE

Step 2) CREATE TEST CASES AND EXECUTE

What do you verify in White Box Testing ?

- Basically verify the **security holes** in the code.
- Verify the **flow of structure** mention in the specification document
- Verify the **Expected outputs**
- Verify the all **conditional loops in the code to check the complete functionality** of the application.
- Verify the **line by line or Section by Section** in the code.

White-box Testing Techniques

The following are some important techniques of white-box testing:

A. Basis Path Testing

B. Structural Testing

C. Logic Based Testing

A. Basis Path Testing

It allows the design and definition of a basis set of execution paths.

The test cases created from the basis path allow the program to be executed in such a way as to examine each possible path through the program by executing each statement at least once.

The following steps can be applied to derive the basis set:

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

B. Structural Testing

The following are some important types of structural testing:

1. Statement coverage testing
2. Branch coverage testing
3. Condition coverage testing
4. Path coverage testing

1. Statement Coverage Testing

Design test cases so that every statement in a program is executed at least once.

2. Branch coverage testing

Test cases are designed such that different branch conditions is given true and false values in turn.

3. Condition coverage testing

Test cases are designed such that each component of a composite conditional expression given both true and false values.

Example

- Consider the conditional expression ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once i.e. given true and false values.

4. Path coverage testing

In path coverage, we write test cases to ensure that each and every path has been traversed at least once.

C. Logic Based Testing

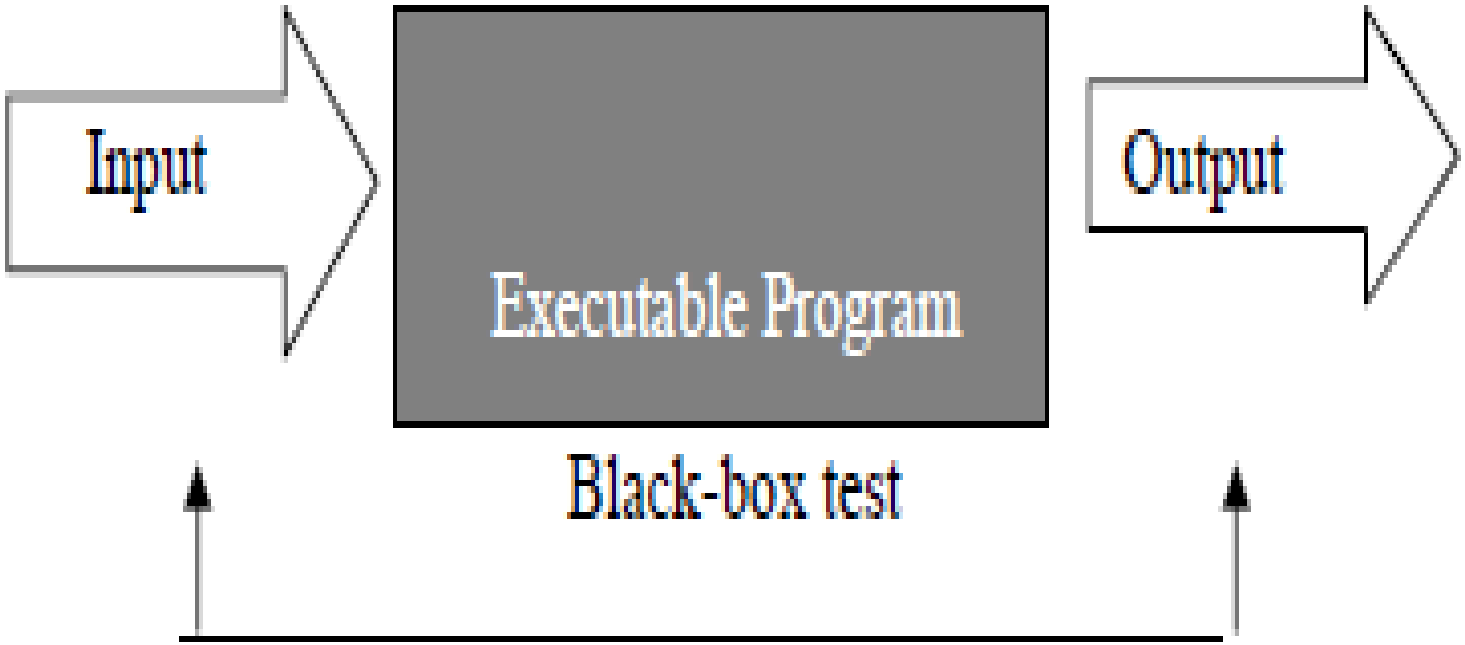
It is used when the input domain and resulting processing are amenable to a **decision table representation.**

- Associate specific conditions with specific actions eliminating impossible combination of conditions.
- Define rules by indicating what action occurs for a set of conditions.

Black Box Testing

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see.

A black-box test takes into account only the input and output of the s/w without regard to the internal code of the program.



BLACK BOX TESTING

- Also called ***functional testing*** *and* ***behavioral testing****.*
- It focuses on determining whether or not a program does what it is supposed to do based on its functional requirements.

It attempts to find errors in the external behavior of the code in the following categories

- incorrect or missing functionality;
- Interface errors;
- Errors in data structures used by interfaces;
- Behavior or performance errors;
- initialization and termination errors.

LEVELS APPLICABLE TO:

Black Box Testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing

How to do Black Box Testing?

Various techniques for the effective Black Box Testing are:

1. Requirements Based Testing
2. Positive and Negative testing
3. Boundary Value Analysis
4. Equivalence Partitioning
5. State/Graph Based Testing

1.Requirements Based Testing

- We begin by looking at each customer requirement, to make sure that every single requirement has been tested at least once.
- Thus, it deals with validating the req. given in the SRS of the s/w system.

- Explicit req. are stated & documented as part of the SRS.
- Whereas Implicit or Implied req. are not documented but assumed to be incorporated in the system.
- Requirements are tracked by a **Requirements Traceability Matrix(RTM)**.

Requirements Traceability Matrix(RTM)

Req. ID	Description	Priority (High, Medium, Low)	Test Conditions	Test Case IDs	Phase of Testing

- Each req. is given a **unique id** along with a brief **description** of it and its **priority**. Next, **Test Conditions** are specified with each id to give a comfort feeling that we have not missed any scenario that could produce a defect. Next, **Test Case Ids** can be used to map b/w req. & Test cases. Next, a **phase of Testing** is to be specified to indicate when a req. will be tested.

2. Positive and Negative Testing

- Positive testing tries to prove that a given product does what is supposed to do. The purpose of positive testing is to prove that the product works as per specification and expectations.
- Negative testing is done to show that the product does not fail when an unexpected input is given. It covers scenarios for which the product is not designed and coded.

3. Boundary Value Analysis(BVA)

- Experience shows that test cases that are close to boundary conditions have a higher chances of detecting an error.
- Here, boundary condition means just below the boundary(upper side) or just above the boundary(lower side).
- Suppose, we have an input variable x with a range from 1-100 . The boundary values are 1,2,99,100.

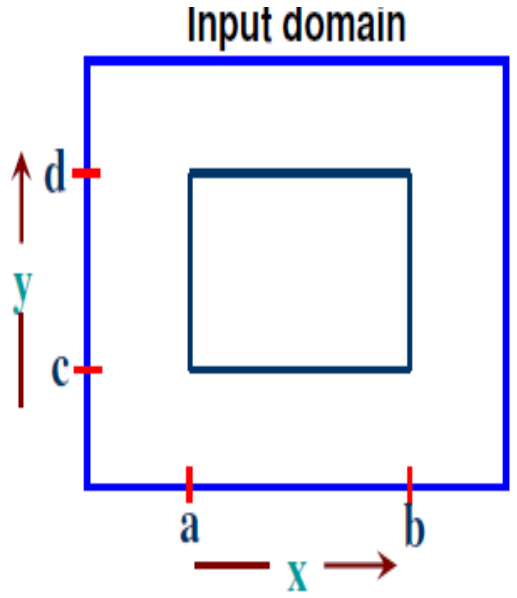
- Consider a program with two input variables x and y. These input variables have specified boundaries as:

  a <= x <= b

  c <= y <= d

  Hence, both the inputs x and y are bounded by two Intervals [a,b] and [c,d] respectively. For input x, we may design test cases with values a and b , just above a and also just below b.

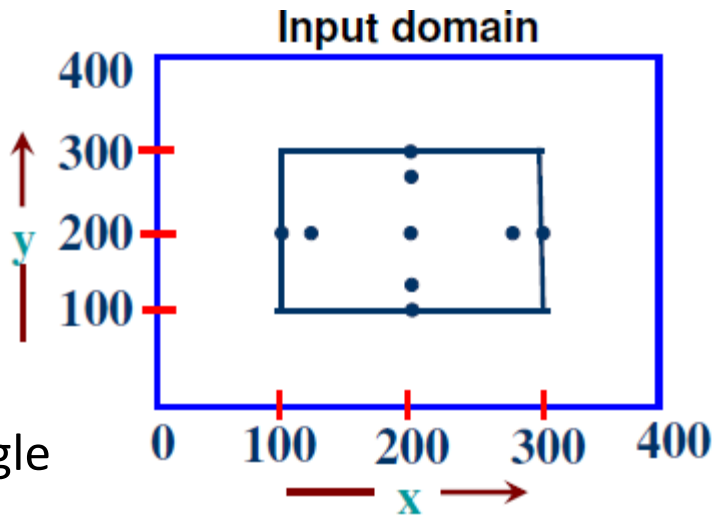
  Similarly, for input y, we may have values c & d, just above c and just below d. These test cases may have more chances to detect an error. The input domain is shown in fig.



Input domain for program having two input variables

The basic idea of boundary value analysis is to use input variable values at their minimum, just above minimum, a nominal value, just below their maximum, and at their maximum.
Thus, boundary value analysis test cases are obtained by **holding the values of all but one variable at their nominal values and letting that variable assume its extreme values.**

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200).
This input domain is shown in Fig. Each dot represent a test case and inner rectangle is the domain of legitimate inputs.



Input domain of two variables x and y with boundaries [100,300] each

Thus, for a program of n variables, boundary value analysis yield 4n + 1 test cases.

Example

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

The boundary value test cases are :

Test Case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example 2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

The boundary value test cases are:

Test Case	*Month*	*Day*	*Year*	*Expected output*
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

4.Equivalence Partitioning

- The set of input values that generate one single expected output is called a **partition**.
- When the behavior of the s/w is the same for a set of values, then the set is termed as an equivalence class or a partition.
- Since all the values produce equal & same output, they are termed as equivalence partition.

E.g. Life Insurance Premium Rates

Age Group	Additional Premium
Under 35	$1.65
35-59	$2.87
60+	$6.00

Based on the age group, an additional monthly premium has to be paid as base premium($0.50) + additional premium.

Now, The equivalence partitions that are based on the age are:

1. Below 35 yrs of age (Valid i/p)
2. Between 35 & 59 yrs of age (Valid i/p)
3. Above 60 yrs of age (Valid i/p)
4. Negative age (Invalid i/p)
5. Age as 0 (Invalid i/p)
6. Age as any three digit number (Invalid i/p)

Example

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Identify the equivalence class test cases for output and input domains.

Output domain equivalence class test cases can be identified as follows:

O_1={<a,b,c>:Not a quadratic equation if a = 0}

O_1={<a,b,c>:Real roots if $(b^2-4ac)>0$}

O_1={<a,b,c>:Imaginary roots if $(b^2-4ac)<0$}

O_1={<a,b,c>:Equal roots if $(b^2-4ac)=0$}`

The number of test cases can be derived form above relations and shown below:

Test case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

Test Case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Example

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

1 <= month <= 12
1 <= day <= 31
1900 <= year <= 2025

The possible outputs would be Previous date or invalid input date.

Identify the equivalence class test cases for output & input domains.

Solution

Output domain equivalence class are:

O_1={<D,M,Y>: Previous date if all are valid inputs}

O_1={<D,M,Y>: Invalid date if any input makes the date invalid}

Test case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

We may have another set of test cases which are based on input domain.

I_1={month: $1 \leq m \leq 12$}

I_2={month: $m < 1$}

I_3={month: $m > 12$}

I_4={day: $1 \leq D \leq 31$}

I_5={day: $D < 1$}

I_6={day: $D > 31$}

I_7={year: $1900 \leq Y \leq 2025$}

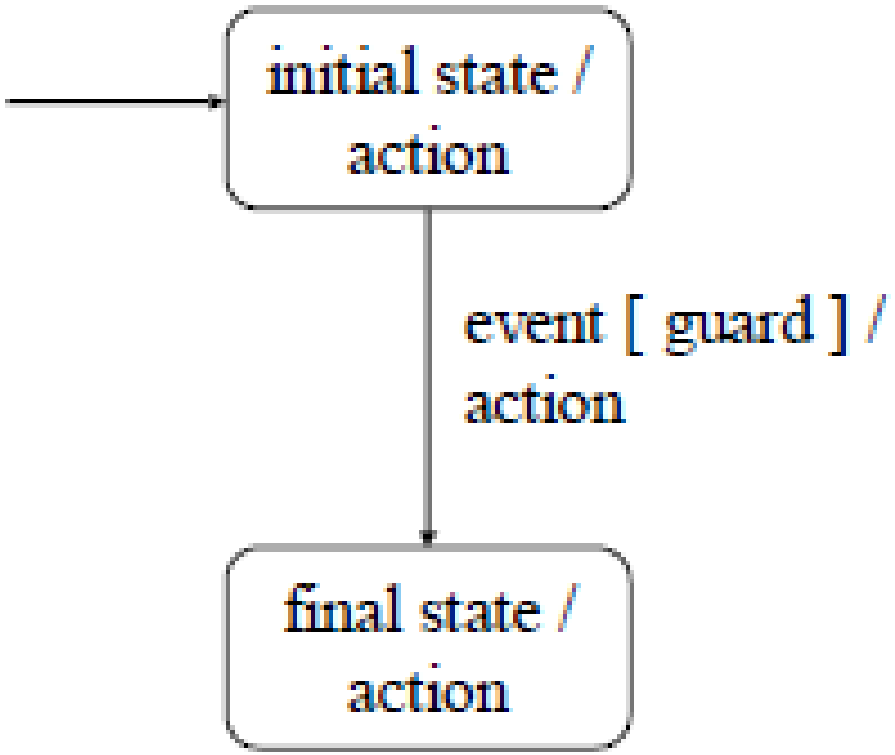
I_8={year: $Y < 1900$}

I_9={year: $Y > 2025$}

Inputs domain test cases are :

Test Case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	invalid input
6	6	32	1962	invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	invalid input (Value out of range)
9	6	15	2026	invalid input (Value out of range)

5. State-based testing



Testing for Functionality

- **System testing begins with function testing**, whereas previous tests concentrates on components and their interaction, this test ignores system structure and focuses on functionality.
- It refers to testing which involves only **observation of the output for certain input values**. There is no attempt to analyze the code which produces the output.
- The internal structure of the code is ignored, therefore it is also referred to as **Black box testing.**

It typically involves five steps:

- The **identification of functions** that the software is expected to perform.
- The **creation of input data** based on the function's specifications.
- The **determination of output** based on the function's specifications.
- The **execution of the test case**.
- The **comparison of actual and expected output**.

Performance Testing

- **Performance testing addresses the non-functional requirements**.
- It is measured against the performance objectives set by the customer as expressed in the non-functional requirements.
- It is **performed to determine how fast some aspect of a system perform under a particular workload**.

Performance Testing

The focus of Performance testing is checking a software program's -

- **Speed** - Determines whether the application responds quickly
- **Scalability** - Determines maximum user load the software application can handle.
- **Stability** - Determines if the application is stable under varying loads

Test Data Suit Preparation

- A **test case is a document** that describes an input, action or event & its expected result, in order to determine whether the software or a part of software is working correctly or not.
- IEEE defines test case as ***"a set of input values, execution preconditions, expected results and execution post conditions developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement."***

Static Testing

- **Static testing** is a form of software testing where the software is not executed; this is in contrast to dynamic testing.
- It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily checking of the code and/or manually reviewing the code or document to find errors.
- This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and Software walkthroughs are also used.

- Bugs discovered at this stage of development are less expensive to fix than later in the development cycle.
- The people involved in static testing are application developers and testers.

Difference between static and dynamic testing

STATIC TESTING	DYNAMIC TESTING
Testing done without executing the program	Testing done by executing the program
This testing does verification process	This testing does validation process
It is about prevention of defects.	It is about finding & fixing the defects
It can be performed before compilation.	It can be performed after compilation.
Cost of finding defects and fixing is less.	Cost of finding defects and fixing is high.
Requires loads of meetings.	Comparatively requires lesser meetings.

Static Testing Strategies

The various static testing strategies are:-

1. Formal Technical Reviews (Peer reviews)
2. Walkthrough
3. Code inspection
4. Compliance with design standards.

Formal Technical Reviews (Peer Reviews)

- Review is "*a process of meeting during which artifacts of software products are examined by project stakeholders, user representatives, or other interested parties for feedback or approval*".
- It can be on technical specifications, design, source code, user documentation, support and maintenance documentation , test plans, test specifications, standards, or any other type specific to the product.
- It can be conducted at any stage of SDLC.
- **Purpose** of conducting review is to minimize the defect ratio as early as possible in SDLC.
- **Informal** reviews are **walkthroughs** & **formal** are **inspection.**

The **objectives** of FTR are:

i. To **uncover** errors in function, logic or implementation for any representation of the software.
ii. To **verify** that the software under review meets its requirements.
iii. To **achieve** software that is developed in a uniform manner.
iv. To **make** projects more manageable.
v. To **ensure** that the software has been represented according to predefines standards.

Walk Through

- Method of conducting informal group/individual review is called walkthrough.
- In this, the participants ask questions & make comments about possible errors, violation of development standards & other problems or may suggest improvement on the article.
- It can be preplanned or can be conducted at need basis .

Code Inspection

- It is a formal group review designed to identify problems as close to their point of origin as possible.
- It is used to improve the quality of product and to improve productivity.

Objectives of Inspection Process

- Find problems at the earliest possible point in the s/w development process.
- Verify that the work product meets its requirements.
- Ensure that the work product has been presented according to predefined standards.
- Provide data on product quality and process effectiveness.