SOFTWARE TESTING

Levels of Testing

Dr. ARIVUSELVAN.K

Associate Professor – (SCORE)

VIT University

UNIT TESTING

Units must be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements.

Though software is divided into modules but a module is not an isolated entity.

While testing the module, if the interfaced modules are not ready, then all its interfaces must be simulated.

Types of interface modules:

(1) Drivers Module:

Used to invoke a module under test:

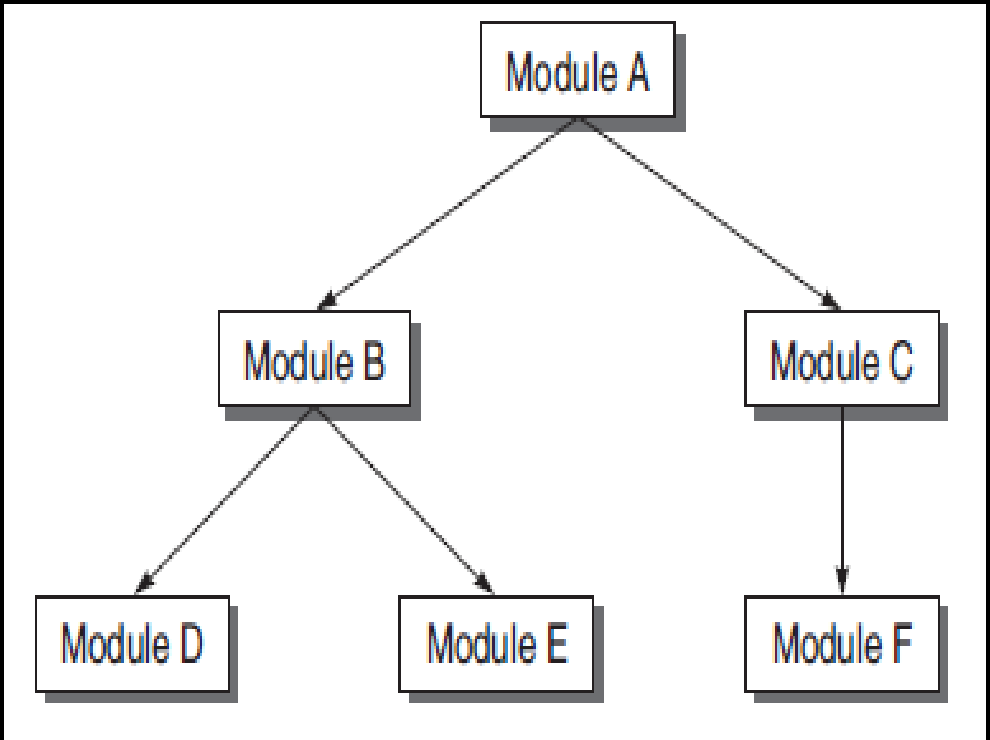
A test driver provides the following facilities to a unit to be tested:

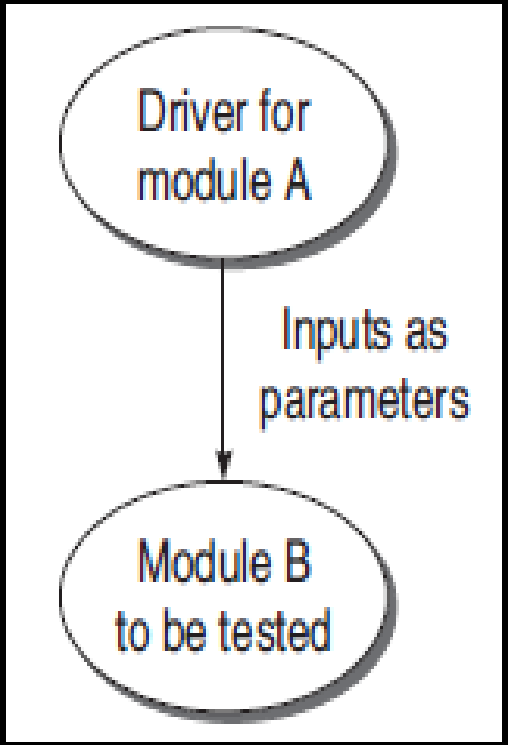
(i) Initializes the environment desired for testing.

(ii) Provides simulated inputs in the required format to the units to be tested.

Example:

Suppose module B is under test, it needs input from module A which is not ready.





Therefore, a driver module is needed which will simulate module A in the sense that it passes the required inputs to module B and acts as a main program for module B.

(2) Stubs Module:

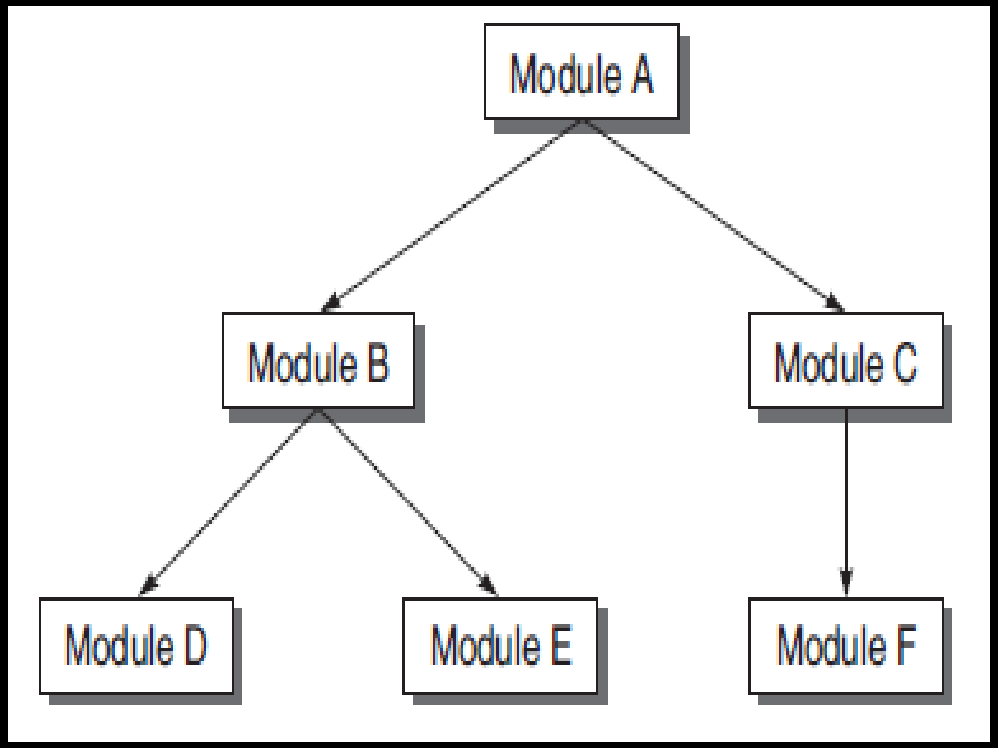
The module under testing may also call some other module which is not ready at the time of testing. Therefore, these modules need to be simulated for testing.

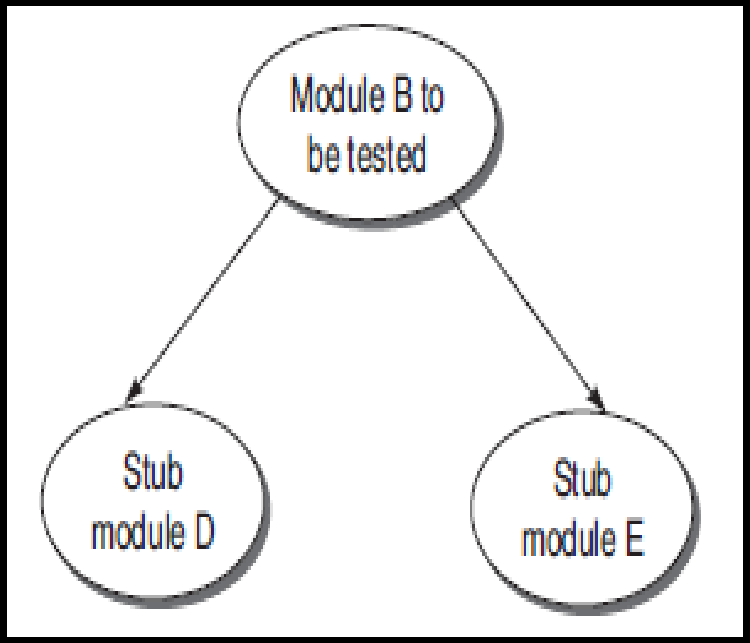
In most cases, dummy modules are prepared for these subordinate modules. These dummy modules are called stubs.

A stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested.

Example:

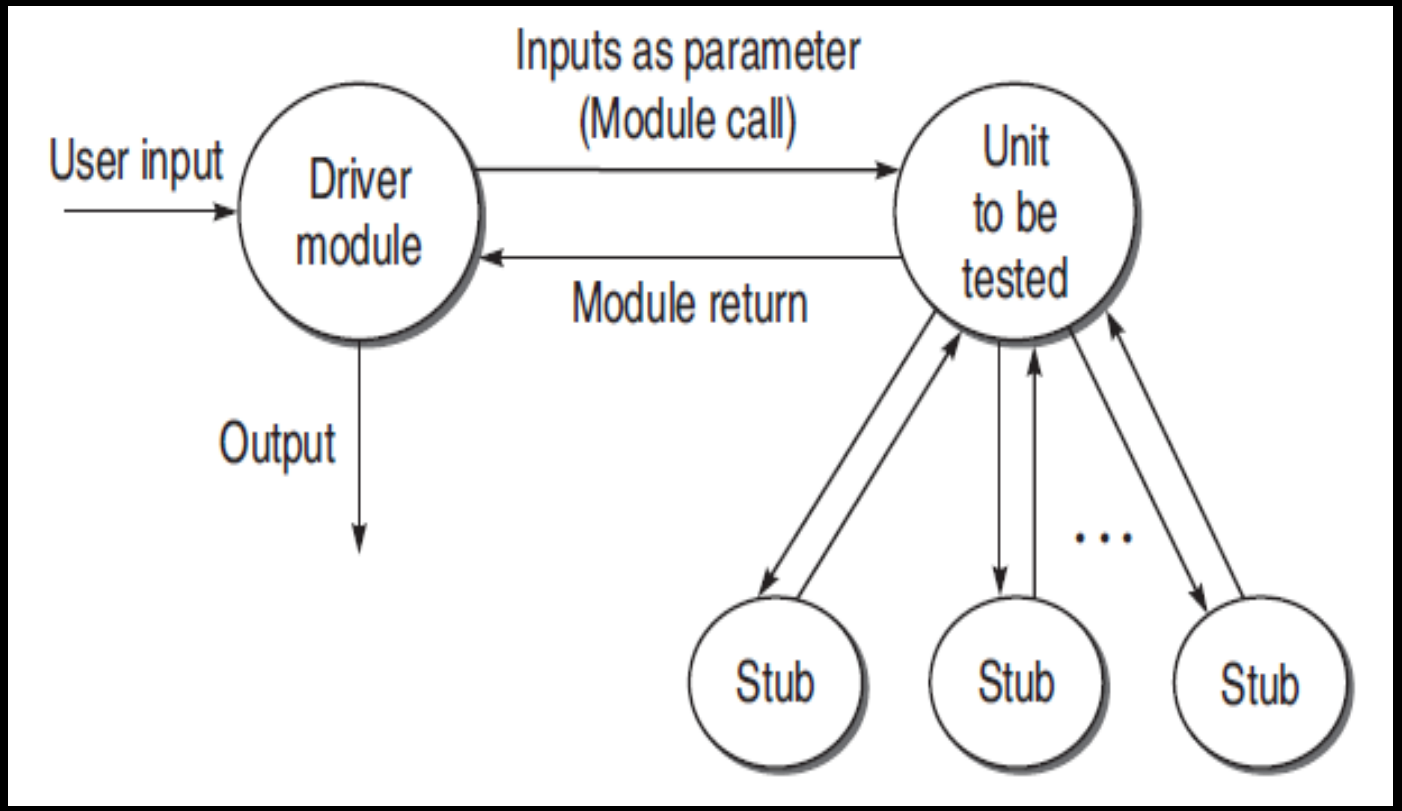
Module B under test needs to call module D and module E. But they are not ready.





Therefore, stubs are designed for module D and module E, as shown in above fig.

Drivers and Stubs



Stubs and drivers are generally prepared by the developer of the module under testing.

Developers use them at the time of unit verification. But they can also be used by any other person who is validating the unit.

Example:1

Consider the following program:

```
main()
{
        int a,b,c,sum,diff,mul;

        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        diff = caldiff(a,b,c);
        mul = calmul(a,b,c);
        printf("%d %d %d", sum, diff, mul);
}

calsum(int x, int y, int z)
{
        int d;

              d = x + y + z;
              return(d);
  }
```

(a) Suppose main() module is not ready for the testing of calsum() module. Design a driver module for main().

(b) Modules caldiff() and calmul() are not ready when called in main(). Design stubs for these two modules.

Solution

(a) Driver for main() module:

```
main()
   {
          int a, b, c, sum;

          scanf("%d %d %d", &a, &b, &c);
          sum = calsum(a,b,c);
          printf("The output from calsum module is %d", sum);
}
```

Solution

(b) Stub for caldiff() Module:

```
caldiff(int x, int y, int z)
{
        printf("Difference calculating module");
        return 0;
}
```

Stub for calmul() Module:

```
calmul(int x, int y, int z)
{
        printf("Multiplication calculation module");
        return 0;
}
```

INTEGRATION TESTING

Why do we need integration testing?

Integration testing is necessary for the following reasons:

(i) Integration testing exposes inconsistency between the modules such as improper call or return sequences.

(ii) Data can be lost across an interface.

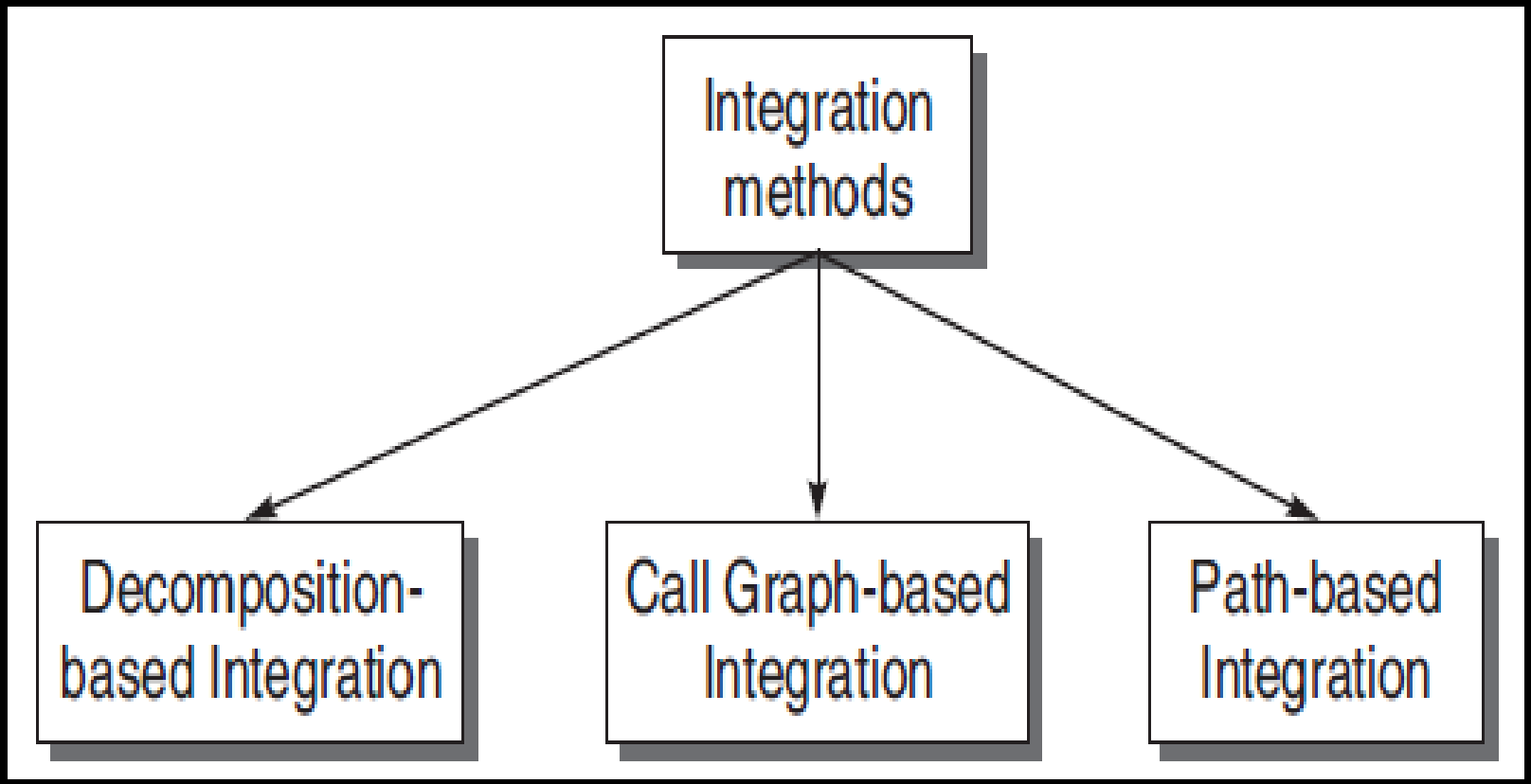
(iii) One module when combined with another module may not give the desired result.

(iv) Data types and their valid ranges may mismatch between the modules.

Integration testing focuses on bugs caused by interfacing between the modules while integrating them.

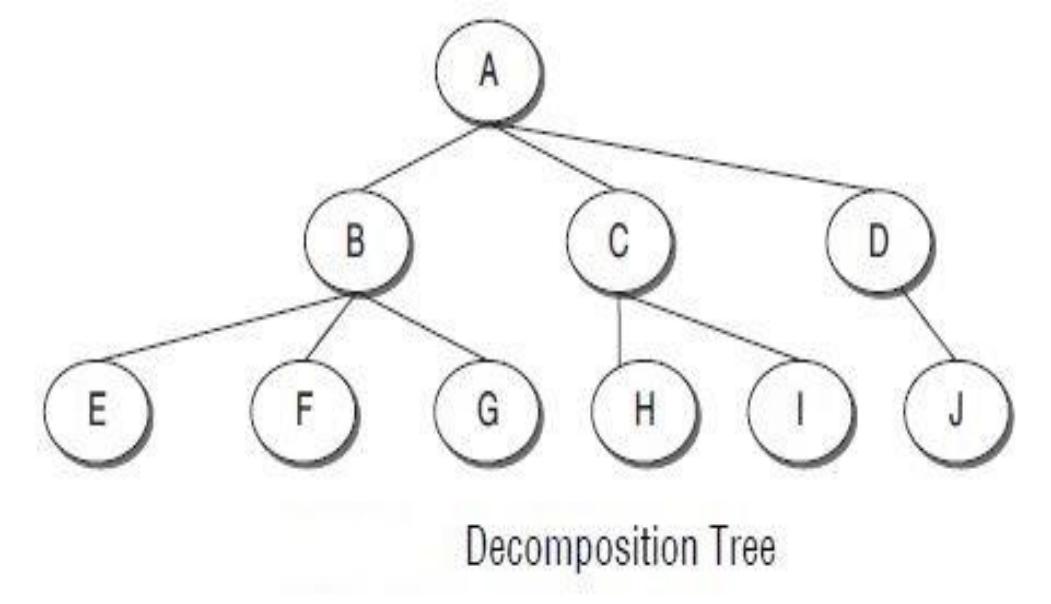
INTEGRATION TESTING

There are three approaches for integration testing:



DECOMPOSITION-BASED INTEGRATION

The idea for this type of integration is based on, the decomposition of design into functional components or modules.



Decomposition Tree

In decomposition-based integration, the nodes represent the modules present in the system.

The links/edges between the two modules represent the calling sequence.

DECOMPOSITION-BASED INTEGRATION

Integration methods in decomposition-based integration depend on two types:

(a) Non-Incremental

(b) Incremental

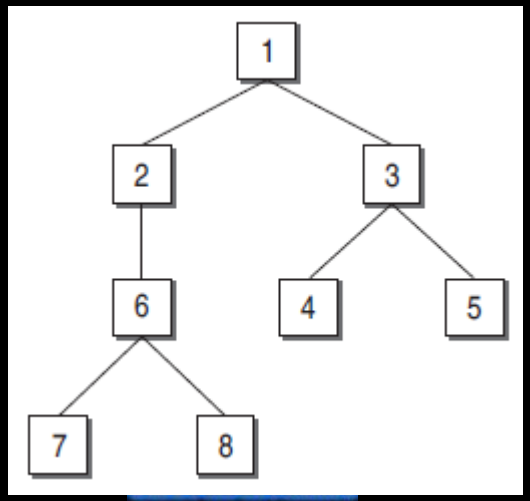
Non-Incremental Integration Testing

In this type of testing, all untested modules are combined together and then tested.

It is also known as Big-Bang integration testing.

Big-Bang method cannot be adopted practically. This theory has been discarded due to the following reasons:

(1) Big-Bang requires more work. For example, consider the following hierarchy of a software system.



Non-Incremental Integration Testing

If all unit tested modules are integrated in this example, then for unit testing of all the modules independently, we require four drivers and seven stubs.

This count will grow according to the size of the system.

Incremental Integration Testing

In this type, you start with one module and unit test it.

Then combine the module which has to be merged with it and perform test on both the modules.

In this way, incrementally keep on adding the modules and test the recent environment.

Types of Incremental Integration Testing

(1) Top-down Integration Testing:

Start with the high-level modules and move downward through the design hierarchy.

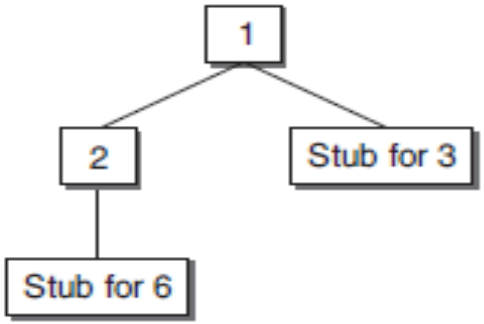
Modules subordinate to the top module are integrated in the following two ways:

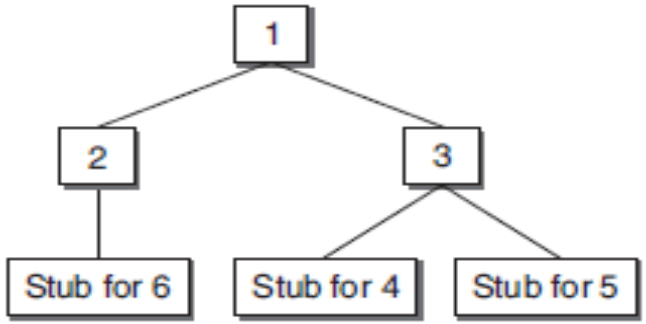
(i) Depth first integration:

In this type, all modules on a major control path of the design hierarchy are integrated first.

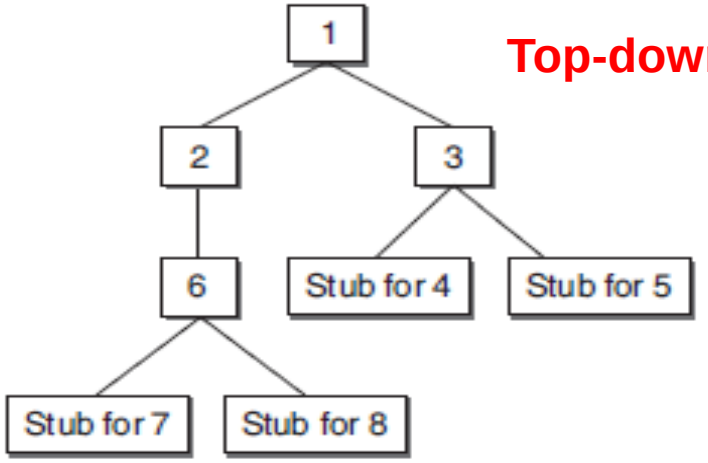
(ii) Breadth first integration :

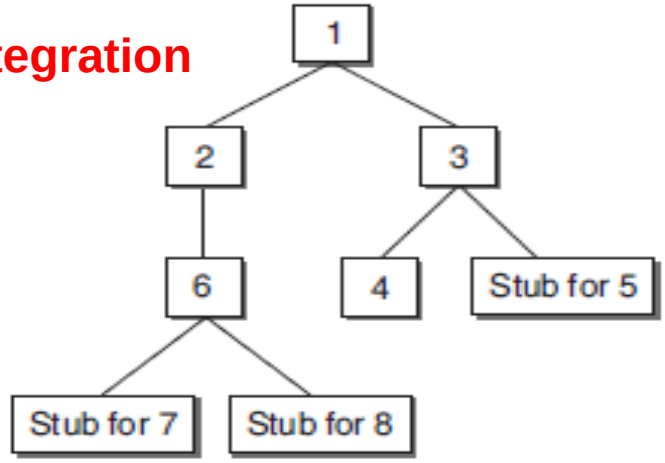
In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first.

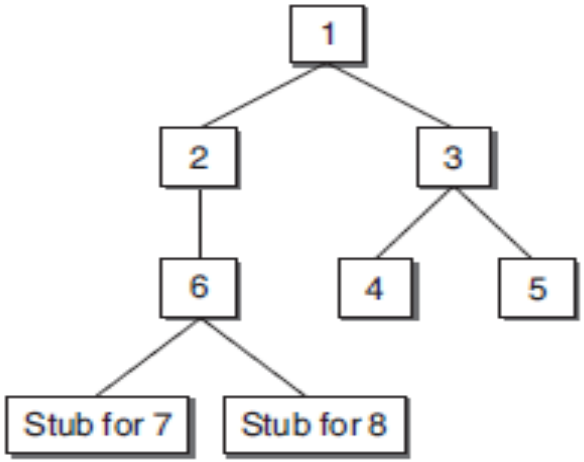


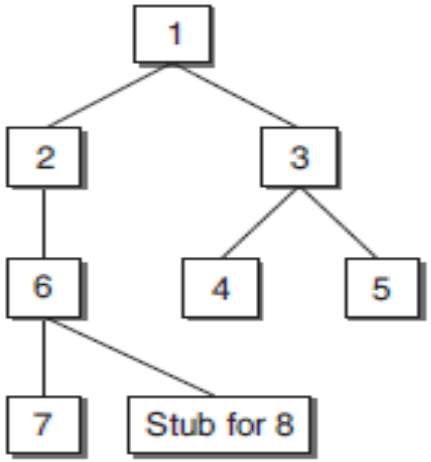


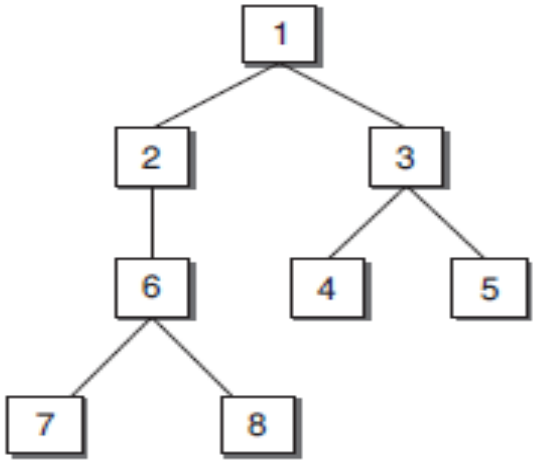
Top-down integration











Types of Incremental Integration Testing

(2) Bottom-up Integration Procedure:

(i) Start with the lowest level modules in the design hierarchy.

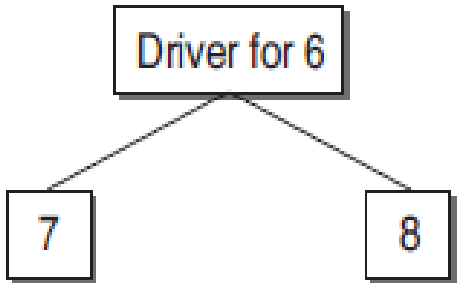
(ii) Look for the super-ordinate module which calls the module selected in step 1. Design the driver module for this super-ordinate module.

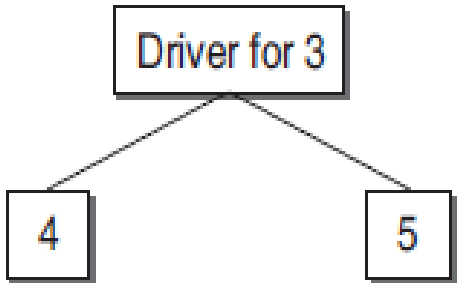
(iii) Test the module selected in step 1 with the driver designed in step 2.

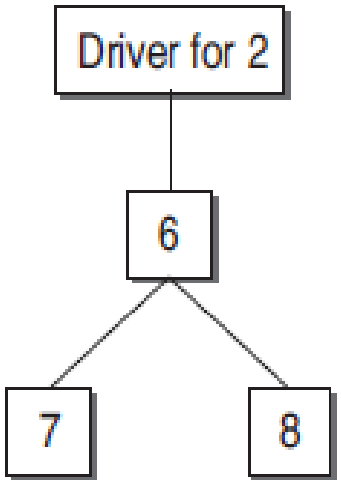
(iv) The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.

(v) Repeat steps 2 to 5 and move up in the design hierarchy.

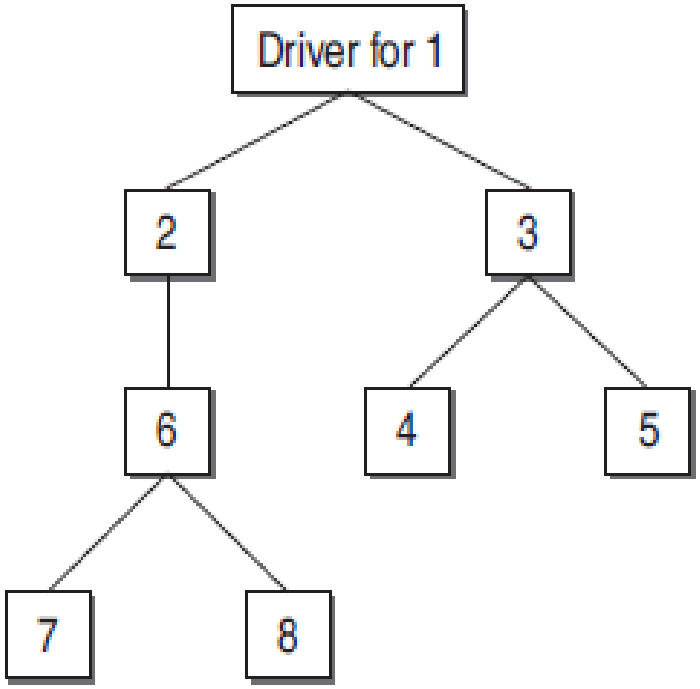
(vi) Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

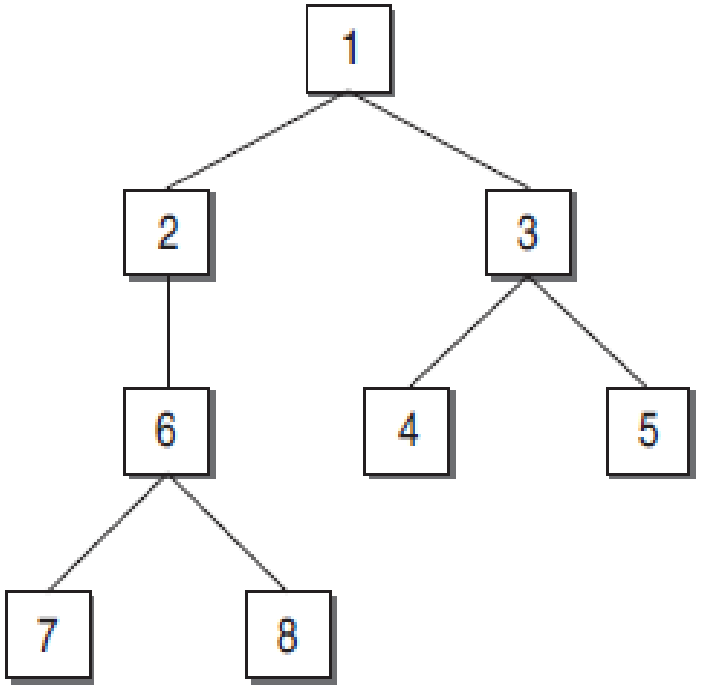






Bottom-up integration





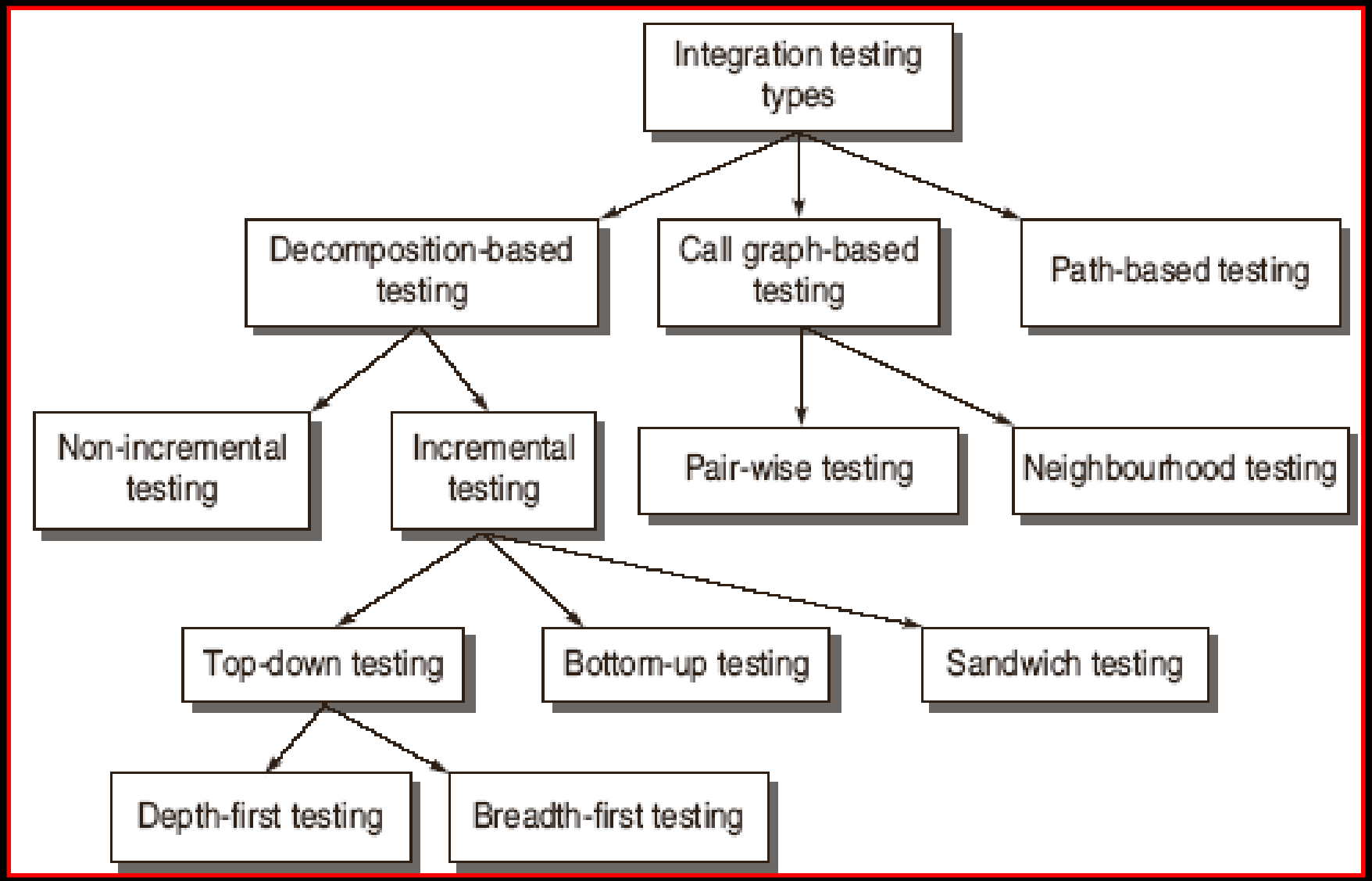
Types of Incremental Integration Testing

Drawbacks of Decomposition-Based Integration:

* More effort required in this type of integration, as stubs and drivers are needed for testing.

* Drivers are more complicated to design as compared to stubs.

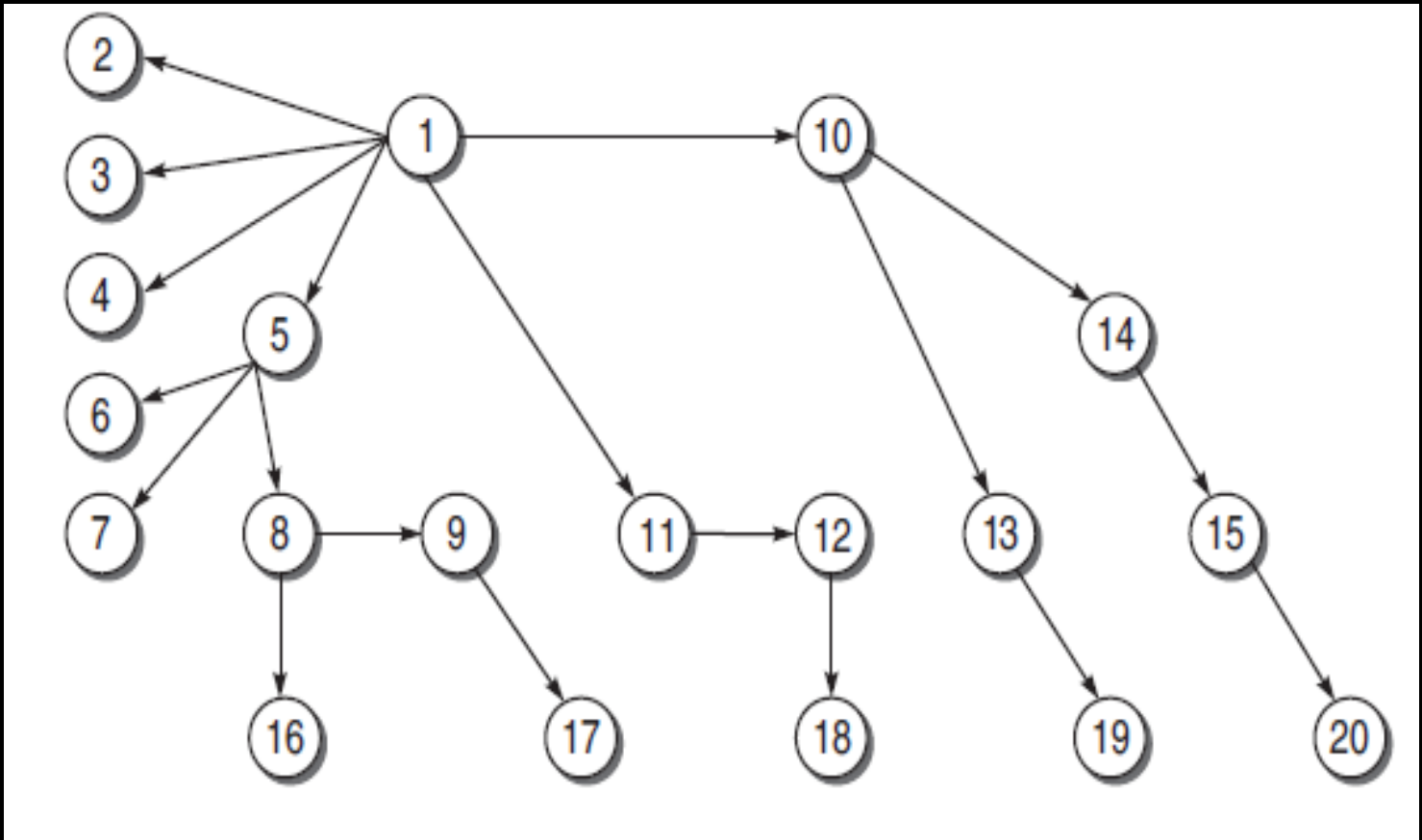
INTEGRATION TESTING TYPES



CALL GRAPH-BASED INTEGRATION

A call graph is a directed graph wherein nodes are modules and a directed edge from one node to another node means one module has called another module.

The call graph can be captured in a matrix form which is known as adjacency matrix.



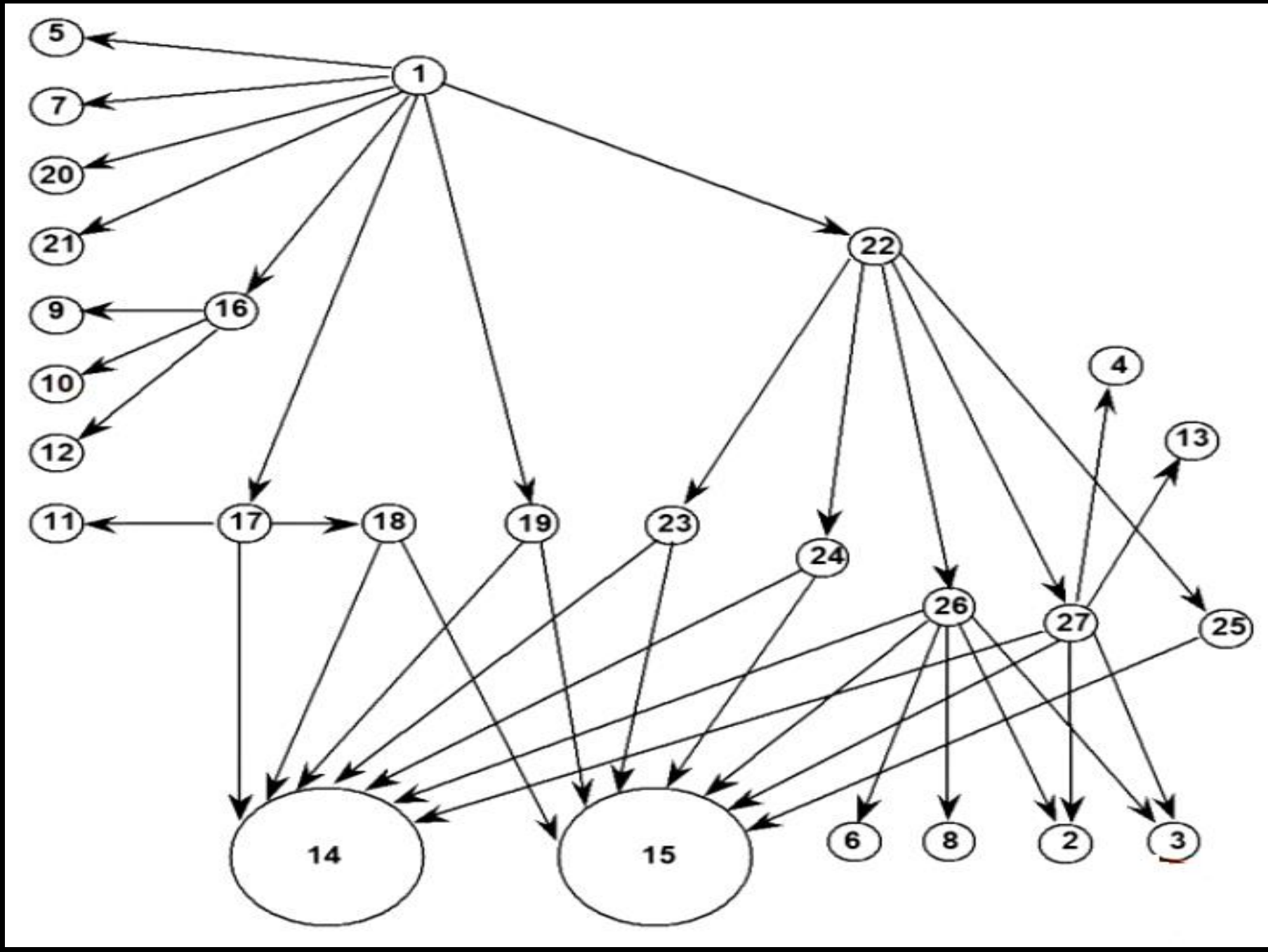
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	x	x	x	x					x	x									
2																			
3																			
4																			
5					x	x	x												
6																			
7																			
8								x							x				
9																x			
10												x	x						
11											x								
12																	x		
13																		x	
14														x					
15																			x
16																			
17																			
18																			
19																			
20																			

Adjacency matrix

Row Sum = Outdegrees

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	×	×	×					×	×									
2																			
3																			
4																			
5					×	×	×												
6																			
7																			
8								×							×				
9																×			
10												×	×						
11											×								
12																	×		
13																		×	
14														×					
15																			×
16																			
17																			
18																			
19																			
20																			

Column Sum = Indegrees



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	Row Sum
1					1		1									1	1		1	1	1	1						8
2																												0
3																												0
4																												0
5																												0
6																												0
7																												0
8																												0
9																												0
10																												0
11																												0
12																												0
13																												0
14																												0
15																												0
16									1	1		1																3
17											1			1				1										3
18														1	1													2
19														1	1													2
20																												0
21																												0
22																							1	1	1	1	1	5
23														1	1													2
24														1	1													2
25															1													1
26		1	1			1		1						1	1													6
27		1	1	1									1	1	1													6
Column Sum	0	2	2	1	1	1	1	1	1	1	1	1	1	7	7	1	1	1	1	1	1	1	1	1	1	1	1	

CALL GRAPH-BASED INTEGRATION

Based on call graph, there are two types of integration testing:

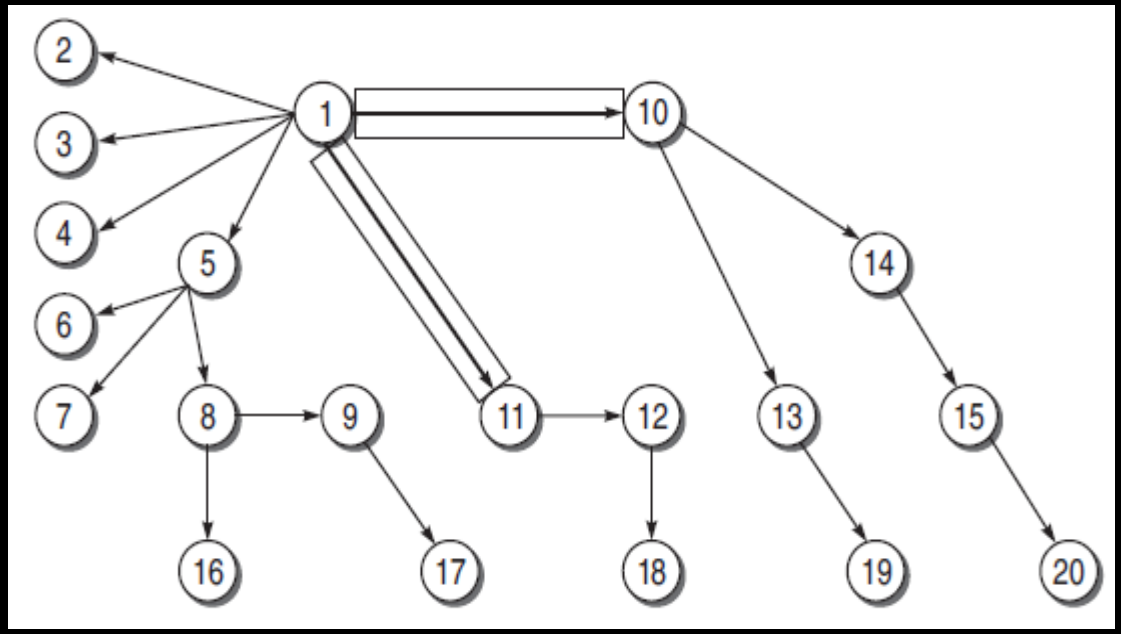
(i) Pair-wise Integration

(ii) Neighborhood Integration

CALL GRAPH-BASED INTEGRATION

(1) Pair-wise Integration:

If we consider only one pair of calling and called modules, then we can make a set of pairs for all such modules,



Resulting Set (Total test sessions) = The sum of all edges in the call graph.

Example : The number of test sessions is 19 which is equal to the number of edges in the call graph.

CALL GRAPH-BASED INTEGRATION

(2) Neighborhood Integration:

The neighborhood of a node, can be defined as the set of nodes that are one edge away from the given node.

The neighborhood for a node is the immediate predecessor as well as the immediate successor nodes.

The neighborhoods of each node in the call graph shown in below Fig.

CALL GRAPH-BASED INTEGRATION

Node	Neighbourhoods	
	Predecessors	Successors
1	-----	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

The total test sessions = Sourcenodes – sink nodes
= 20 -10
= 10

Where, sink node is an instruction in a module at which the execution terminates.

PATH-BASED INTEGRATION

Passing of control from one unit to another unit is necessary for integration testing.

There should be information within the module regarding:

=> instructions that call the module (or)

=> return to the module.

It can be done with the help of path-based integration.

PATH-BASED INTEGRATION

Important terms for path-based integration:

(1) Source node: It is an instruction in the module at which the execution starts or resumes. (or)
The nodes where the control is being transferred after calling the module are also source nodes.

(2) Sink node: It is an instruction in a module at which the execution terminates.
The nodes from which the control is transferred are also sink nodes.

(3) Module execution path (MEP) : It is a path consisting of a set of executable statements within a module like in a flow graph.

(4) Message: When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as a message.

PATH-BASED INTEGRATION

(5) MM-path: It is a path consisting of MEPs and messages.

The path shows the sequence of executable statements; it also crosses the boundary of a unit when a message is followed to call another unit.

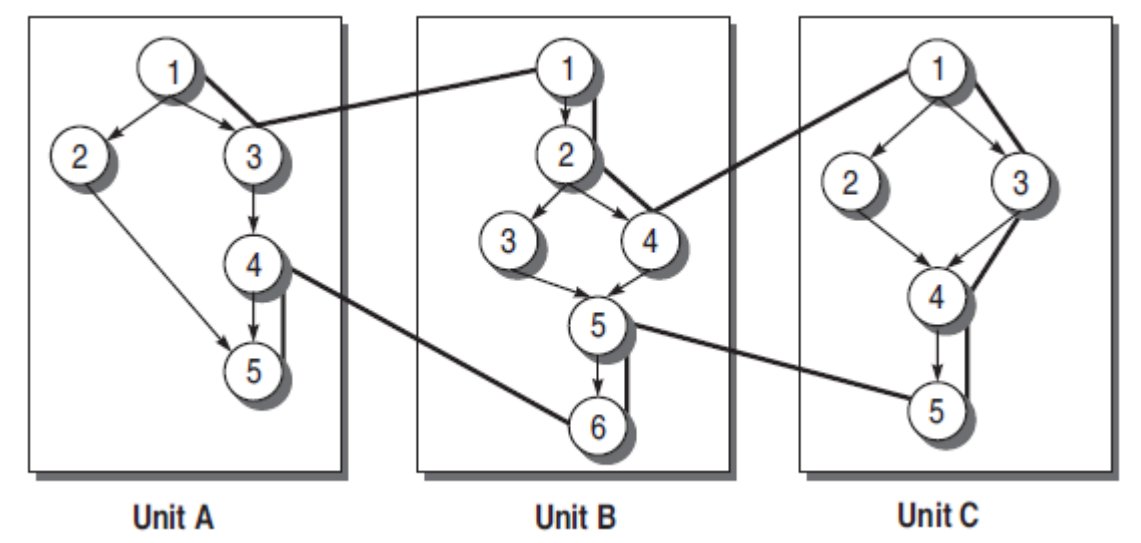
(6) MM-path graph: It can be defined as an extended flow graph where nodes are MEPs and edges are messages.

* It returns from the last called unit to the first unit where the call was made.

* In this graph, messages are highlighted with thick lines.

PATH-BASED INTEGRATION

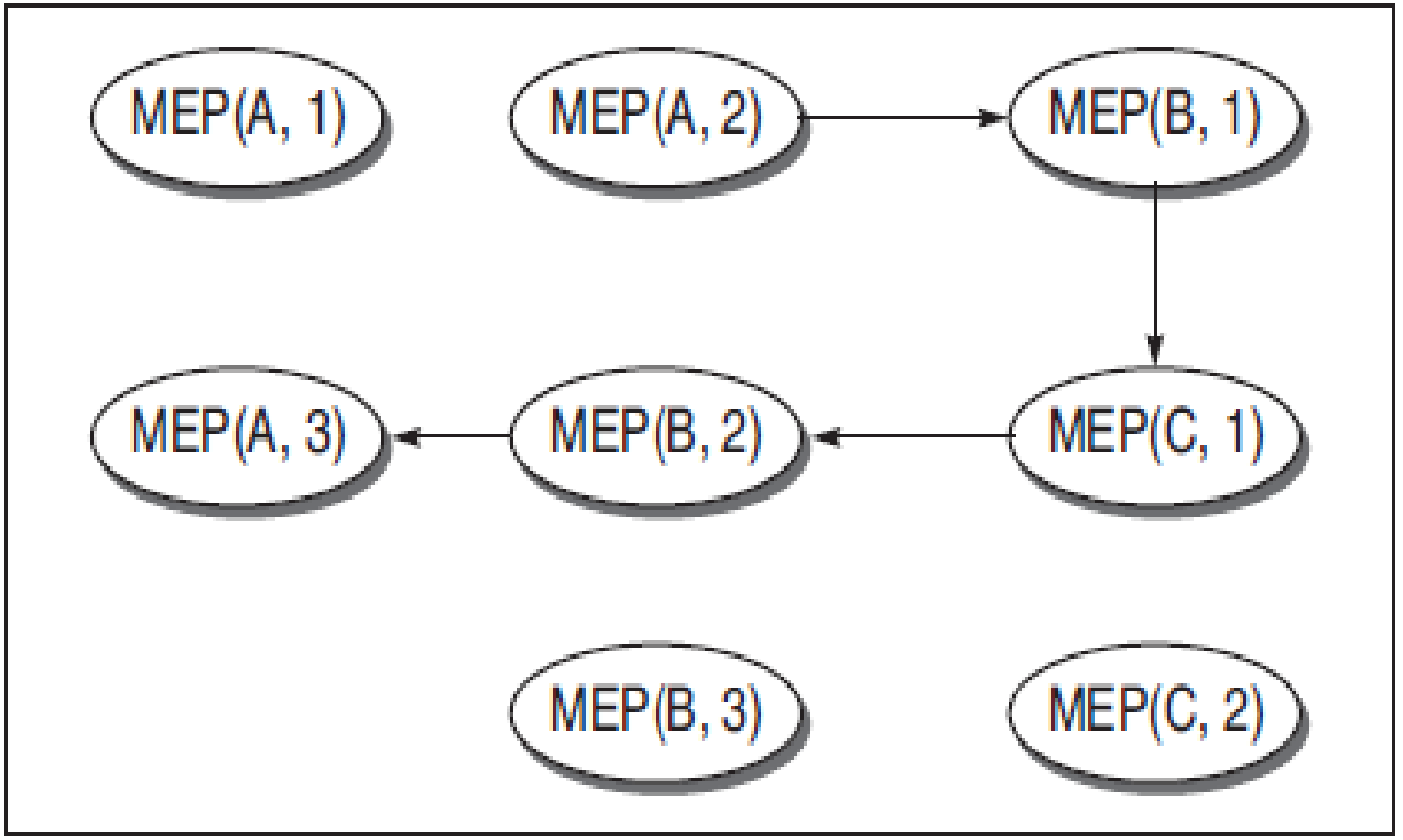
Example:



MM - PATH

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

MM – PATH Details



Module execution path (MEP) Graph

Function Testing

The process of attempting to detect discrepancies between the functional specifications of a software and its actual behavior.

The function test must determine if each component or business event:

- performs in accordance to the specifications,

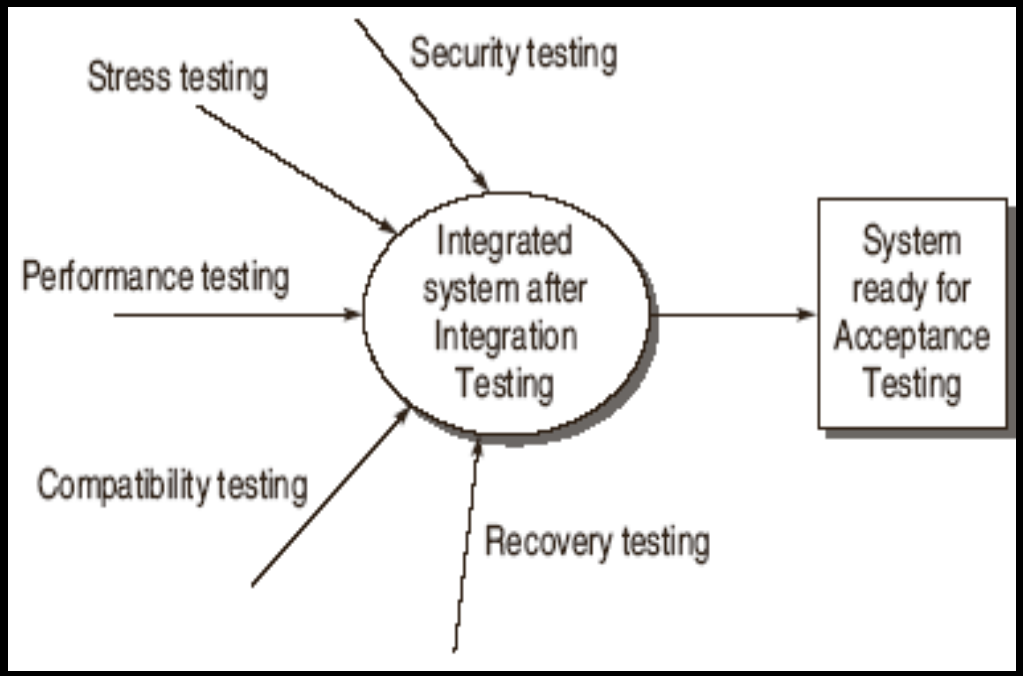
- responds correctly to all conditions that may be presented by incoming events / data.

System Testing

A series of different tests to test the whole system on various grounds where bugs have the probability to occur.

The ground can be performance, security, maximum load, etc.

After passing through these tests, the resulting system is a system which is ready for acceptance testing.



Recovery Testing

Recovery is just like the exception handling feature in a programming language

Recovery is the ability of a system to restart operations after the integrity of the application has been lost.

It reverts to a point where the system was functioning correctly and then reprocesses the transactions up until the point of failure.

Security Testing

Security is a protection system that is needed to assure the customers that their data will be protected.

Elements of security testing:

- **Confidentiality**
- **Integrity**
- **Authentication**
- **Authorization**
- **Non-repudiation**

Performance Testing

(i) Load Testing: When a system is tested with a load that causes it to allocate its resources in maximum amounts, it is called load testing.(i.e. simulates expected user loads)

(ii) Stress Testing: Stress testing tries to break the system under test by overwhelming its resources in order to find the circumstances under which it will crash. (i.e. pushes a system beyond its normal capacity)

Usability Testing

Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.

- **Ease of Use**
- **Interface steps**
- **Response Time**
- **Help System**

Acceptance Testing

Determine whether the software is fit for the user to use.

Making users confident about product.

Enable the buyer to determine whether to accept the system.

- Alpha Testing
- Beta Testing

Alpha Testing

(i) Alpha Testing: It is performed at the development site, testers and users together perform this testing.

Alpha testing is typically done for two reasons:

- **To give confidence that the software is in a suitable state to be seen by the customers (but not necessarily released).**

- **To find bugs that may only be found under operational conditions.**

Beta Testing

(i) Beta Testing: The test period during which the product should be complete and usable in a production environment.

The software is released to groups of people so that further testing can ensure the product has few or no bugs.

Sometimes, beta-versions are made available to the open public to increase the feedback to a maximal number of future users.

Versions of the software, known as beta-versions, are released to a limited audience outside the company.

Regression Testing

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Regressive Testing

(1) **Baseline version:** The version of a component (system) that has **passed a test suite.**

(2) **Delta version:** A **changed** version that has **not passed** a regression test.

(3) **Delta build:** An **executable configuration** of the **SUT** (System Under Test) that contains all the **delta and baseline** components.

Regression testing is **not another testing** activity. Rather, it is the **re-execution** of **some or all** of the **already developed** test cases.

REGRESSION TESTING PRODUCES QUALITY SOFTWARE