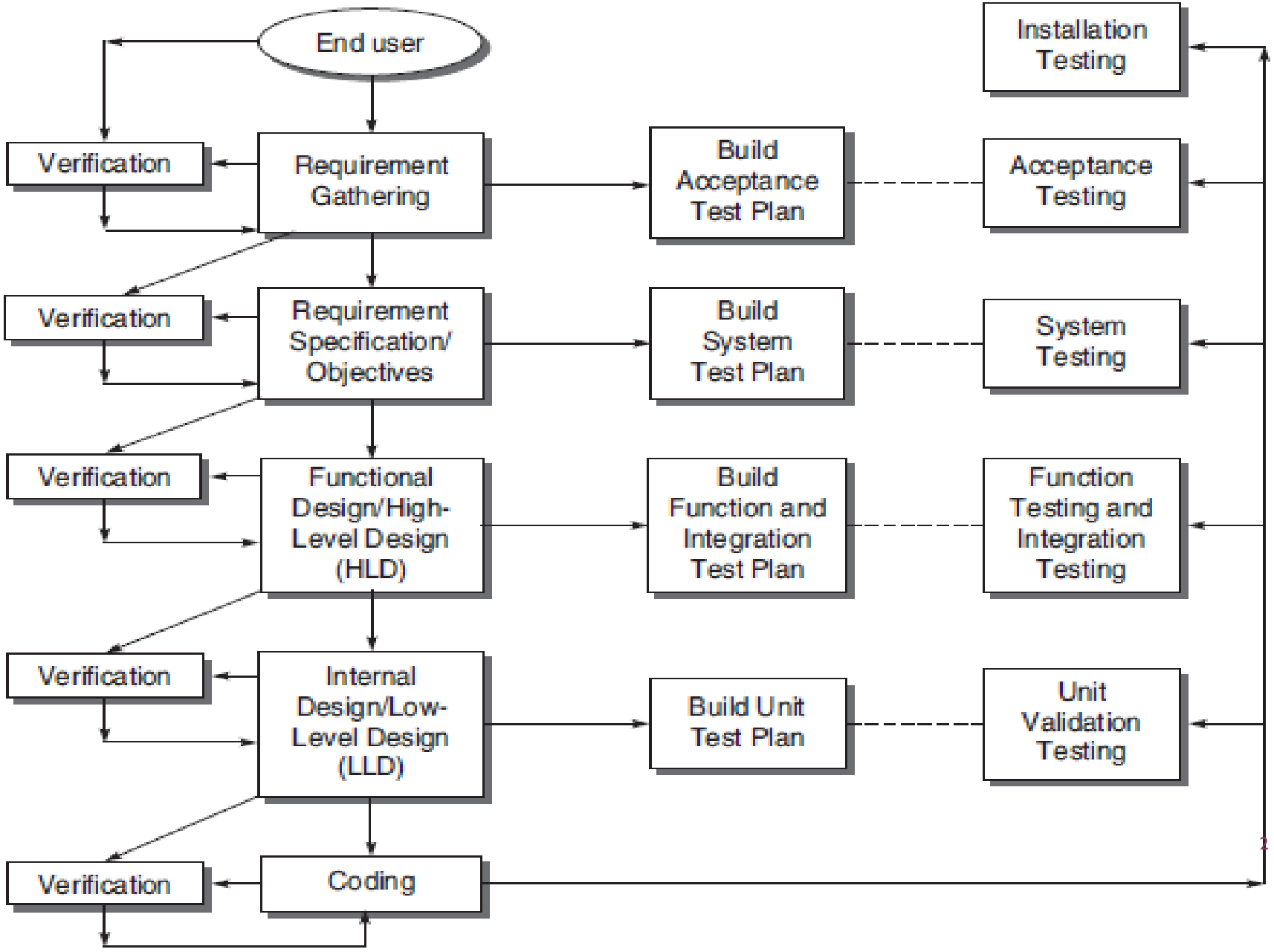
UNIT:2 TYPES OF TESTING

VALIDATION ACTIVITIES:

V & V ACTIVITIES:



UNIT VALIDATION TESTING:

- Unit is the smallest building block of the software system, it is the first piece of system to be validated.
- Before we validate the entire software, units or modules must be validated.
- Unit testing is normally considered an adjunct to the coding step.
- Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements.
- Unit tests ensure that the software meets at least a **baseline level** of functionality prior to integration and system testing.

UNIT VALIDATION TESTING:

- While developing a software, if developing the software, if the developer detects and removes the bug, it offers significant savings of time and costs.
- This type of testing is largely based on black-box techniques.
- Though software is divided into modules but a module is not an isolated entity. The module under consideration might be getting some inputs from another module or the module is calling some other module.
- It means that a module is not independent and cannot be tested in isolation.
- While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration.

[1] DRIVERS:

- Suppose a module is to be tested, wherein some inputs are to be received from another module.
- However, this module which passes inputs to the module to be tested is not ready and under development.
- In such a situation, we need to simulate the inputs required in the module to be tested.
- For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test.
- This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a ***driver module.***

DRIVERS:

- Therefore, it can be said that a test driver is supporting the code and data used to provide an environment for testing a part of a system in isolation.
- In fact, it drives the unit being tested by creating necessary 'inputs' required for the unit and then invokes the unit.
- A test driver may take inputs in the following form and call the unit to be tested:

1. It may hard-code the inputs as parameters of the calling unit.
2. It may take the inputs from the user.
3. It may read the inputs from a file.

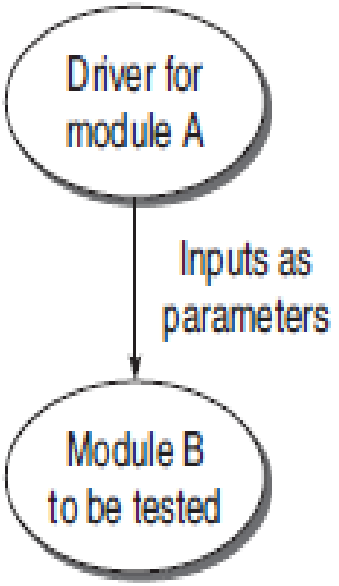


Figure 7.3 Driver Module for module A

[2] STUBS:

- 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 instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called ***stubs***.
- Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit.

STUBS:

- Stubs have the following characteristics:

1. Stub is a place holder for the actual module to be called. Therefore, it is not designed with the functionalities performed by the actual module. It is a reduced implementation of the actual module.
2. It does not perform any action of its own and returns to the calling unit (which is being tested).
3. We may include a display instruction as a trace message in the body of stub. The idea is that the module to be called is working fine by accepting the input parameters.
4. A constant or null must be returned from the body of stub to the calling module.
5. Stub may simulate exceptions or abnormal conditions, if required.

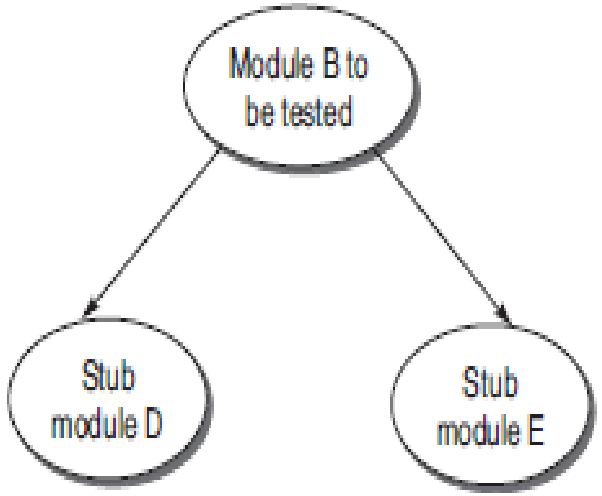


Figure 7.4 Stubs

EXAMPLE:

INTEGRATION TESTING:

- In the modular design of a software system where the system is composed of different modules, integration is the activity of combining the modules together when all the modules have been prepared.
- Integration aims at constructing a working software system.
- Why do we need integration testing? When all modules have been verified independently, then why is integration testing necessary? As discussed earlier, modules are not standalone entities. They are a part of a software system which comprises of many interfaces. Even if a single interface is mismatched, many modules may be affected.

INTEGRATION TESTING:

- Integration testing is necessary for the following reasons:

1. Integration testing exposes inconsistency between the modules such as improper call or return sequences.
2. Data can be lost across an interface.
3. One module when combined with another module may not give the desired result.
4. Data types and their valid ranges may mismatch between the modules.

INTEGRATION TESTING:

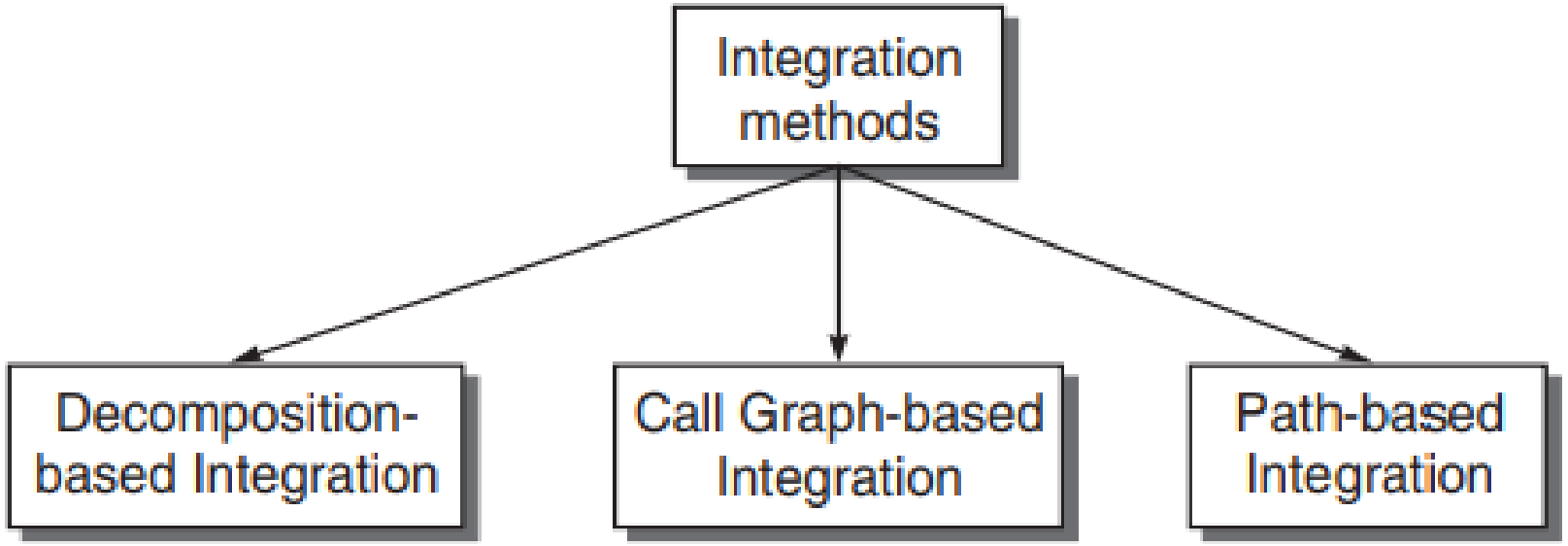


Figure 7.6 Integration Methods

CALL GRAPH BASED INTEGRATION:

- It is assumed that integration testing detects bugs which are structural. However, it is also important to detect some behavioural bugs.
- If we can refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioural testing at the integration level.
- This can be done with the help of a call graph, as given by Jorgensen.
- A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- The call graph can be captured in a matrix form which is known as the adjacency matrix.

EXAMPLE:

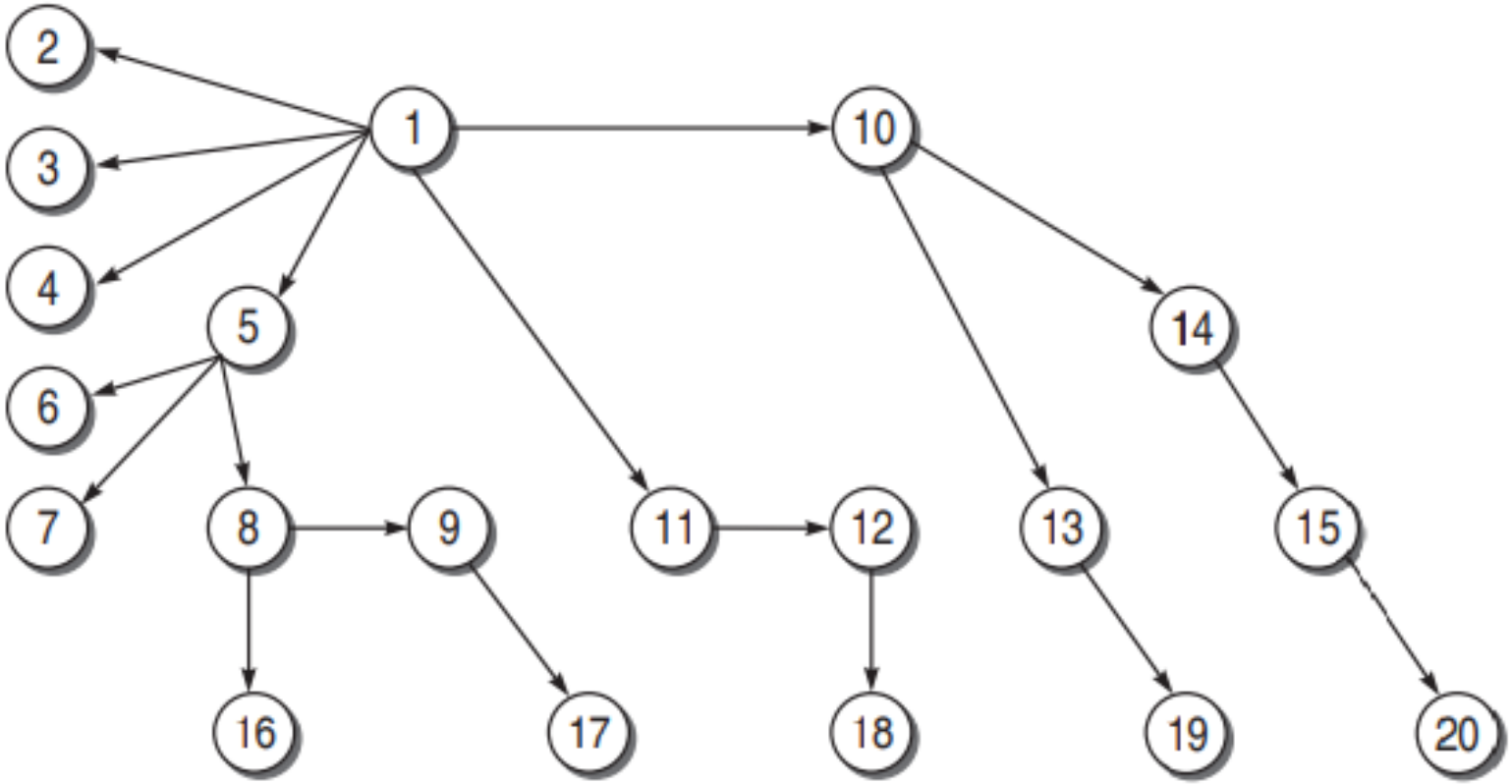


Figure 7.9 Example call graph

EXAMPLE:

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																			

Figure 7.10 Adjacency matrix

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, as shown in Fig. 7.11 for pairs 1–10 and 1–11.
- The resulting set will be the total test sessions which will be equal to the sum of all edges in the call graph.
- For example, in the call graph shown in Fig. 7.11, the number of test sessions is 19 which is equal to the number of edges in the call graph.

PAIR-WISE INTEGRATION:

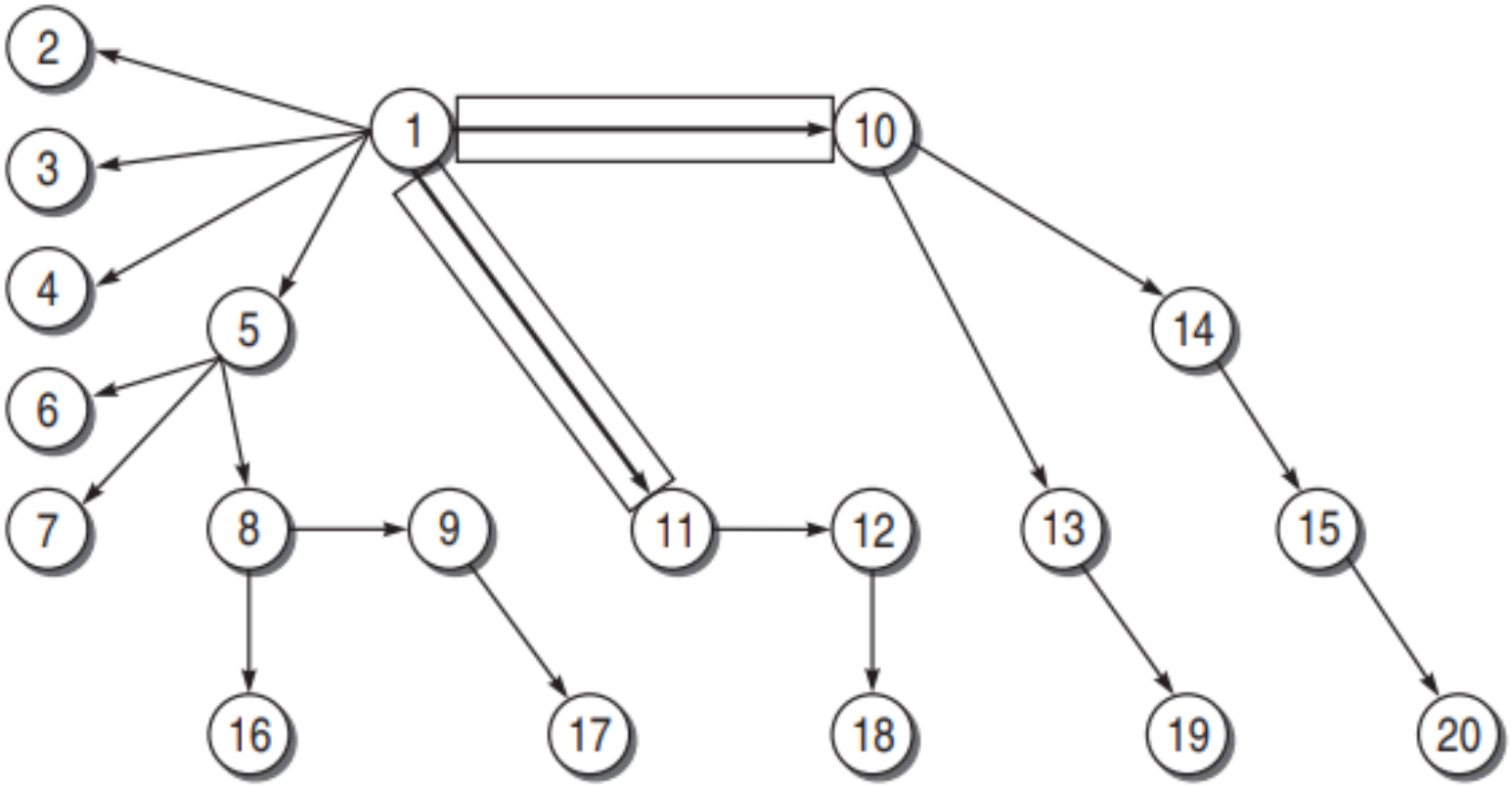


Figure 7.11 Pair-wise integration

NEIGHBOURHOOD INTEGRATION:

- There is not much reduction in the total number of test sessions in pair-wise integration as compared to decomposition-based integration.
- If we consider the neighbourhoods of a node in the call graph, then the number of test sessions may reduce.
- The neighbourhood for a node is the immediate predecessor as well as the immediate successor nodes.
- The neighbourhood of a node, thus, can be defined as the set of nodes that are one edge away from the given node.

NEIGHBOURHOOD INTEGRATION:

Table 7.2 Neighbourhood integration details

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 in neighbourhood integration can be calculated as:
Neighbourhoods = nodes – sink nodes

= 20 – 10 = 10

where sink node is an instruction in a module at which the execution terminates

7.2.3 PATH-BASED INTEGRATION

- It may be possible that in that path execution, there may be a call to another unit. At that point, the control is transferred from the calling unit to the called unit.
- This passing of control from one unit to another unit is necessary for integration testing.
- Also, there should be information within the module regarding instructions that call the module or return to the module.
- This must be tested at the time of integration. It can be done with the help of path-based integration defined by Paul C. Jorgenson [78].

PATH-BASED INTEGRATION:

- **Source node** It is an instruction in the module at which the execution starts or resumes. The nodes where the control is being transferred after calling the module are also source nodes.
- **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.
- **Module execution path (MEP)** It is a path consisting of a set of executable statements within a module like in a flow graph.
- **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.
- **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.
- **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.

7.3 FUNCTION TESTING

- When an integrated system is tested, all its specified functions and external interfaces are tested on the software.
- Every functionality of the system specified in the functions is tested according to its external specifications.
- An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user).
- ***Function testing as the process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.***

FUNCTION TESTING:

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

1. performs in accordance to the specifications,
2. responds correctly to all conditions that may present themselves by incoming events/data,
3. moves data correctly from one business event to the next (including data stores), and
4. is initiated in the order required to meet the business objectives of the system.

FUNCTION TESTING:

- The primary processes/deliverables for requirements based function test are discussed below.

1. Test planning
2. Partitioning/functional decomposition
3. Requirement definition
4. Test case design
5. Traceability matrix formation
6. Test case execution

7.4 SYSTEM TESTING

- ***System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirement specification.***
- System testing is actually 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.
- The integrated system is passed through various tests based on these grounds and depending on the environment and type of project.
- After passing through these tests, the resulting system is a system which is ready for acceptance testing which involves the user, as shown in Fig. 7.14.

SYSTEM TESTING:

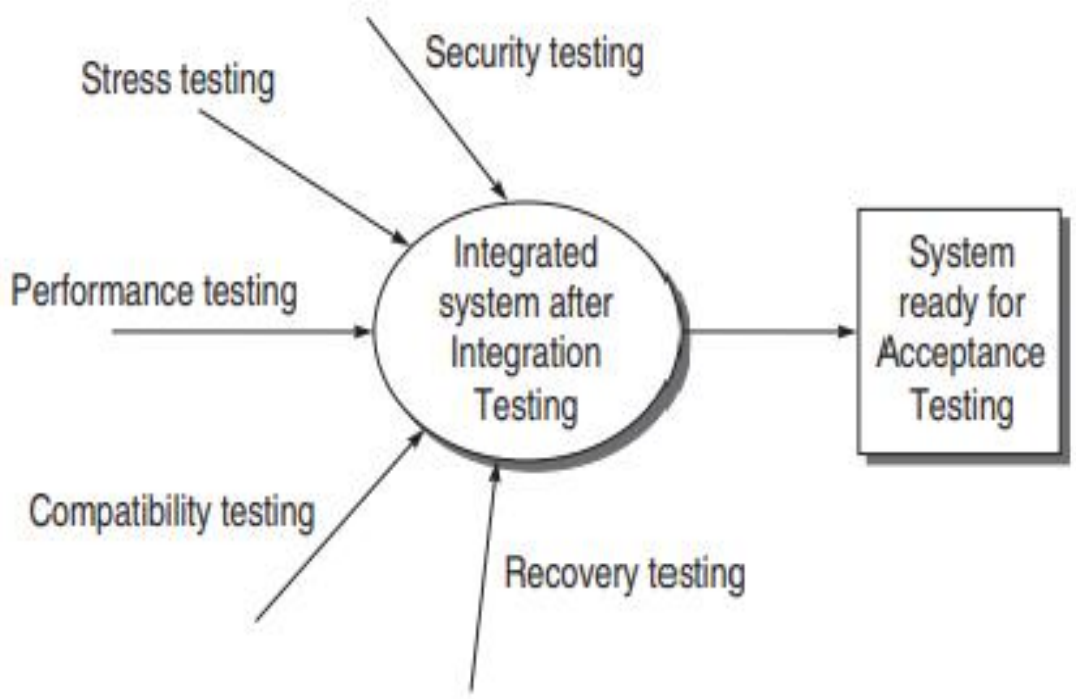


Figure 7.14 System testing

- Different Tests can be performed during System Testing are:

1. Security Testing
2. Stress Testing
3. Performance Testing
4. Compatibility Testing
5. Recovery Testing

7.5 ACCEPTANCE TESTING

- Developers/testers must keep in mind that the software is being built to satisfy the user requirements and no matter how elegant its design is, it will not be accepted by the users unless it helps them achieve their goals as specified in the requirements.
- ***Acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.***
- Acceptance testing must take place at the end of the development process.
- It consists of tests to determine whether the developed system meets the predetermined functionality, performance, quality, and interface criteria acceptable to the user.
- Therefore, the final acceptance acknowledges that the entire software product adequately meets the customer's requirements.

ACCEPTANCE TESTING:

- Thus, acceptance testing is designed to:

1. Determine whether the software is fi t for the user.
2. Making users confident about the product.
3. Determine whether a software system satisfies its acceptance criteria.
4. Enable the buyer to determine whether to accept the system or not.

ACCEPTANCE TESTING:

- Entry Criteria:

1. System testing is complete and defects identified are either fixed or documented.
2. Acceptance plan is prepared and resources have been identified.
3. Test environment for the acceptance testing is available.

- Exit Criteria:

1. Acceptance decision is made for the software.
2. Acceptance decision is made for the software.

TYPES ACCEPTANCE TESTING:

Alpha Testing

- Alpha testing is conducted at developer's end.
- It is performed in the environment controlled by developer.
- It is performed before software is released to the end user.
- The developer keeps record of all errors and bugs.
- It involved Black-box and White-box testing.

Beta Testing

- Beta testing is conducted at customer's end.
- It is performed in the environment not controlled by developer.
- It is performed after software is released to the end user.
- The end user keeps record of all errors and bugs.
- It involved only Black-box testing.

ASSIGNMENT:

1. What is unit testing? Discuss the role of Drivers and Stubs.
2. Write a short note on Call graph-based integration.
3. Explain Path based integration in detail.
4. What is function testing? Explain primary processes of it.
5. What is system testing? Explain its categories in brief.
6. What is acceptance testing? Explain its types.
7. Write a short note on alpha testing.
8. Write a short note on beta testing.
9. Give difference between alpha and beta testing.

MCQ's

7.1 to 7.12, 7.15, 7.19 to 7.24


Thank
you