SWE 4603

Software Testing and Quality Assurance

Lecture 5 Part 1

Prepared By Maliha Noushin Raida, Lecturer, CSE
Islamic University of Technology

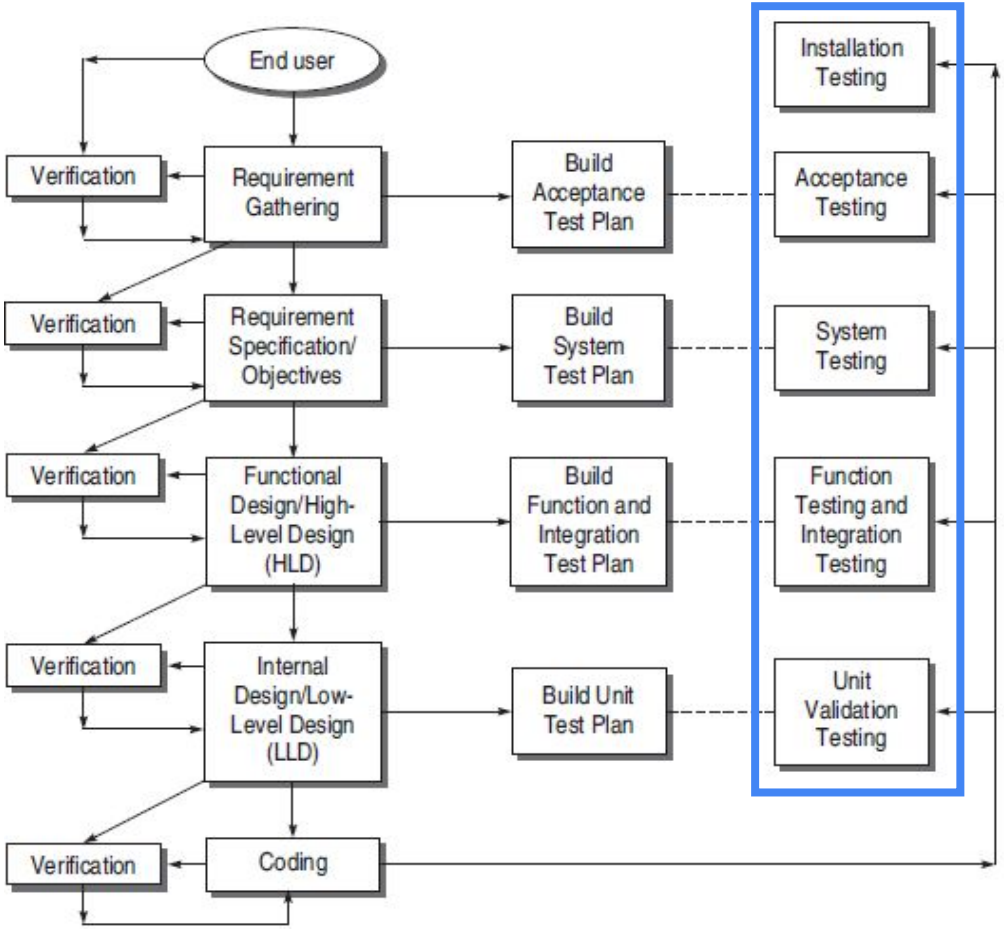
Lesson Outcome

- Unit Testing
- Integration Testing

Week 5 On:

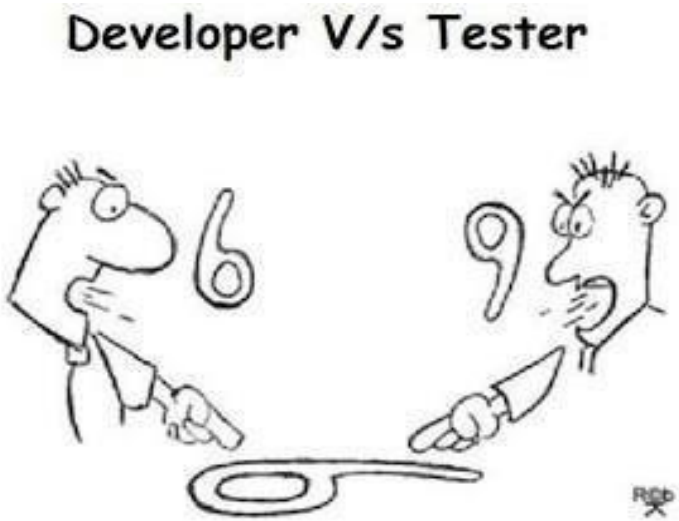
- Chapter 7: Validation Activities

Validation Activities



Unit Validation Testing

Unit Testing



Unit Testing helps increase our confidence in our code

“If it isn’t tested, assume it doesn’t work”

Unit Testing

A *unit test* is a piece of code that invokes a unit of work and checks one specific end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed.

Unit testing is understood as testing small parts of the code in isolation.

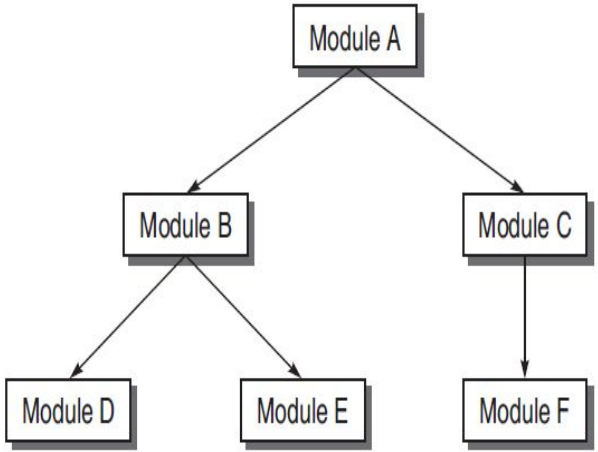
The unit under consideration might be getting some inputs from another unit or the unit is calling some other unit. It means that a unit is not independent and cannot be tested in isolation.

While testing a unit, all its interfaces must be simulated if the interfaced units are not ready at the time of testing the unit under consideration.

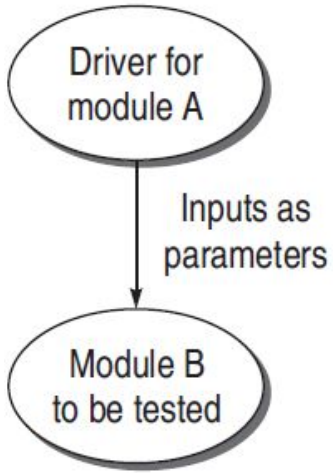
Things to know

Driver:

- Sometimes unit to be tested need input form other unit/module and that unit may not be implemented or under development
- In such a situation, we need to simulate the inputs required in the module to be tested.
- 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.



Suppose module A is not ready and B has to be unit tested. In this case, module B needs inputs from module A. Therefore, a **driver module** is needed which will simulate module A in the sense that it passes the required inputs to module B



Things to know

A test driver may take inputs in the following form and call the unit to be tested:

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

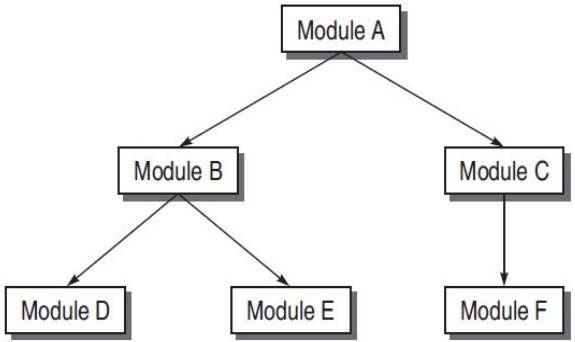
A test driver provides the following facilities to a unit to be tested:

- Initializes the environment desired for testing.
- Provides simulated inputs in the required format to the units to be tested.

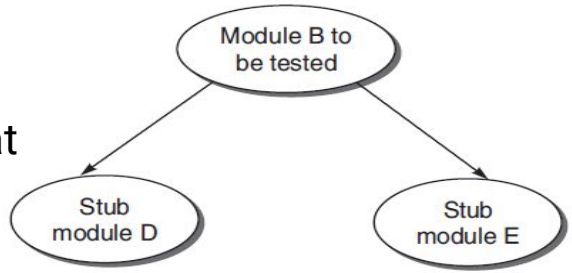
Things to know

Stubs:

- The module under testing may also call some other module which is not ready at the time of testing.
- these modules also need to be simulated for testing.
- For this dummy modules are prepared which is known as stubs.



Module B under test needs to call module D and module E. But they are not ready. So there must be some skeletal structure in their place so that they act as dummy modules in place of the actual modules. Therefore, **stubs** need to be designed for module D and module E



Example

```
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.

Example(Continues)

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);
}
```

Stud for *caldiff()* and *calmul()* 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;
}
```

How to define a **Unit**?

- It’s entirely up to us!
- There are no hard and fast rules for this.
- We can consider a bunch of functions as one unit or test a single function as a separate unit.

But We can use a very simple rule:

If a unit proves difficult to test, then it’s very likely that it needs to be broken down into smaller components.

Writing Good Tests

- Goal: to expose problems!
  - ✔ Assume role of an adversary
  - ✔ Failure == success
- Test boundary conditions
  - ✔ 0, Integer.MAX_VALUE, empty array
- Test different categories of input
  - ✔ positive, negative, and zero
- Test different categories of behavior
  - ✔ each menu option, each error message
- Test “unexpected” input
  - ✔ null pointer, last name includes a space
- Test representative “normal” input
  - ✔ random, reasonable values

Excuses For not Testing

- It takes too long to run the tests
- It's not my job to test my code
- I don't really know how the code is supposed to behave so I can't test it
- But it compiles!
- I'm being paid to write code, not to write tests
- I feel guilty about putting testers and QA staff out of work
- My company won't let me run unit tests on the live system

The Big Excuse

Tests are expensive to write

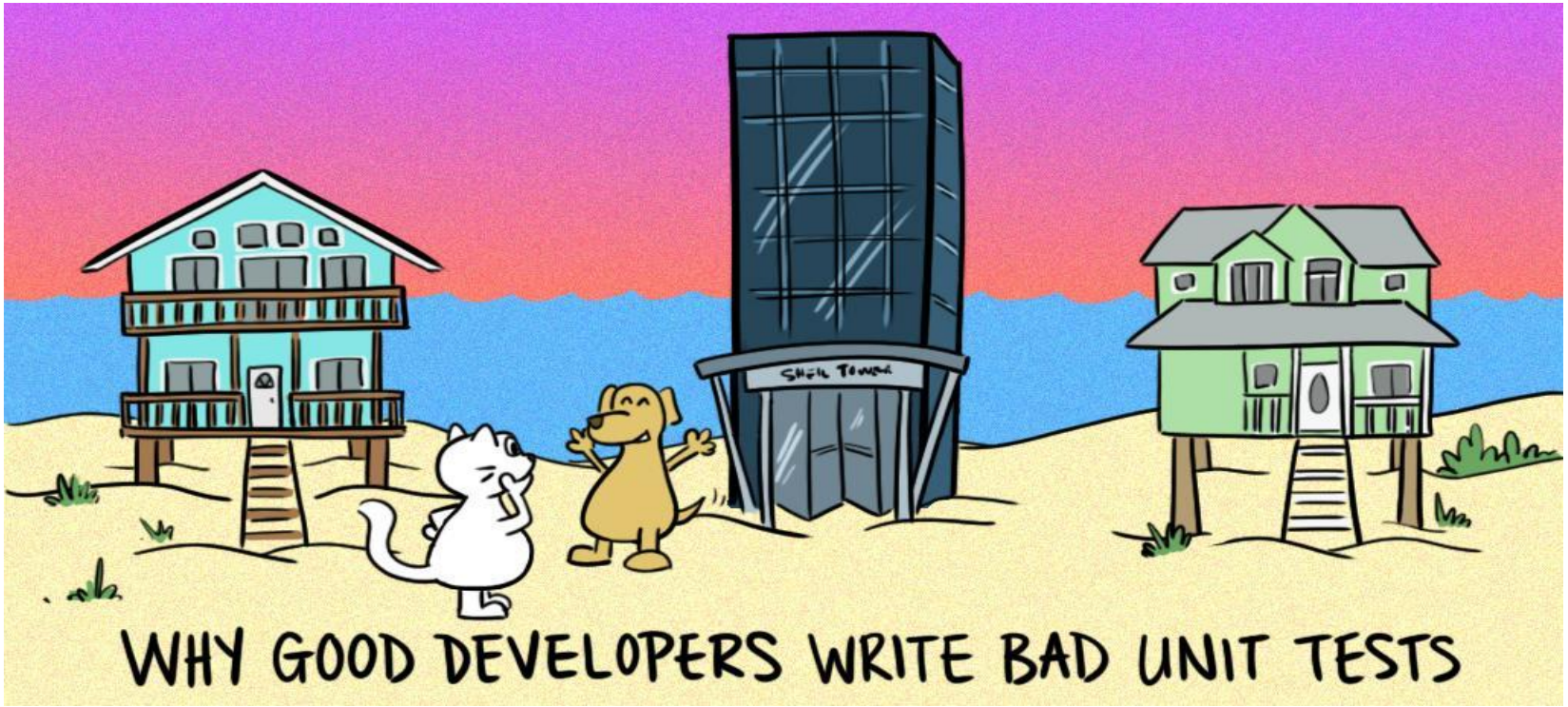
Yes, testing is expensive in most of industries: think about testing a home appliance, a drug or a new car.

But **code is incredibly cheap**, giving the impression that tests are needlessly costly, in a relative way. They do **require extra effort**, but they are efficient complement or even replacement for specifications, they improve quality, bring fast feedback, secure knowledge for newcomers.

Unit Test Best Practices

- Unit Test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
- Test only one code at a time.
- Follow clear and consistent naming conventions for your unit tests
- In case of a change in code in any module, ensure there is a corresponding unit Test Case for the module, and the module passes the tests before changing the implementation
- Bugs identified during unit testing must be fixed before proceeding to the next phase in SDLC

Why good developers write bad unit test?



Utilize 15 mins to read : https://mtlynch.io/good-developers-bad-tests/

Integration Testing

- In **integration testing** smaller units are integrated into larger units and larger units into the overall system.
- This differs from unit testing in that units are no longer tested independently but in groups, the focus shifting from the individual units to the interaction between them.
- The purpose of integration testing is to verify the functional, performance, and reliability between the modules that are integrated.

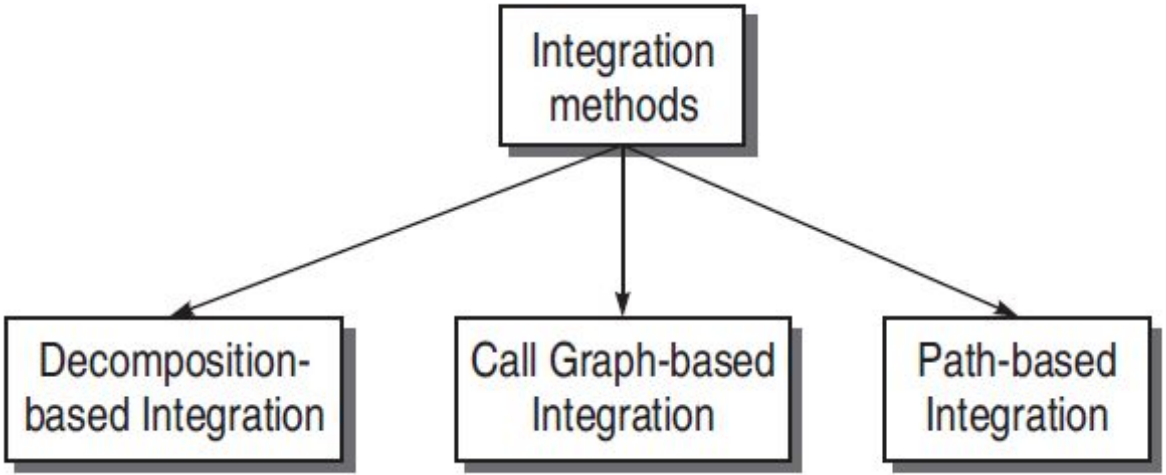
Integration Testing

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.

Types of Integration Testing

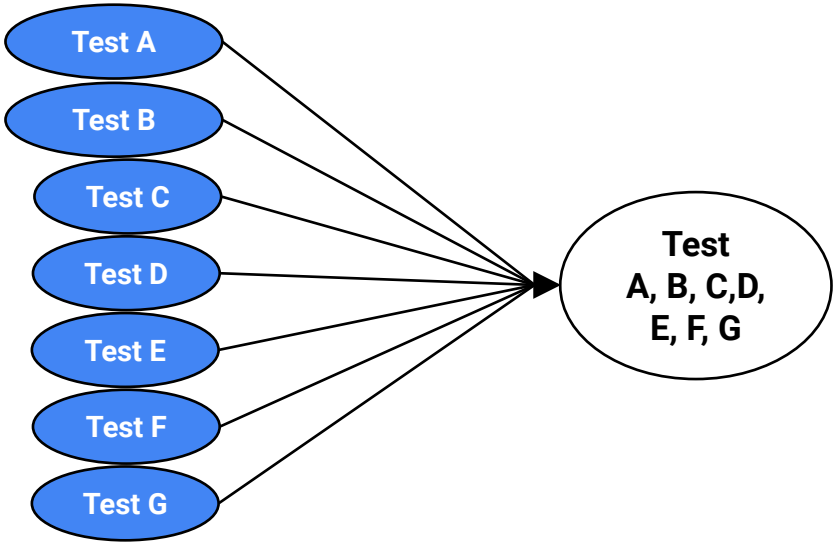
There are three approaches for integration testing.

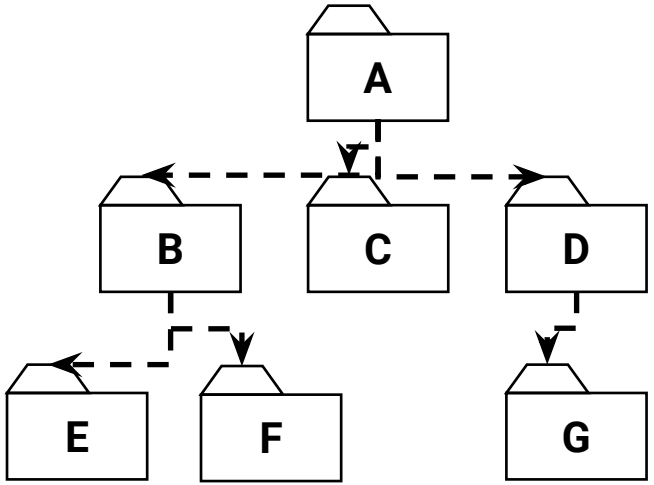


Integration Testing: Decomposition Based

Big-Bang Approach

- This unit tests each of the subsystems, and then does one massive integration test, in which all the subsystems are immediately tested together.
- Don‘t try this!! Why: The interfaces of each of the subsystems have not been tested yet.

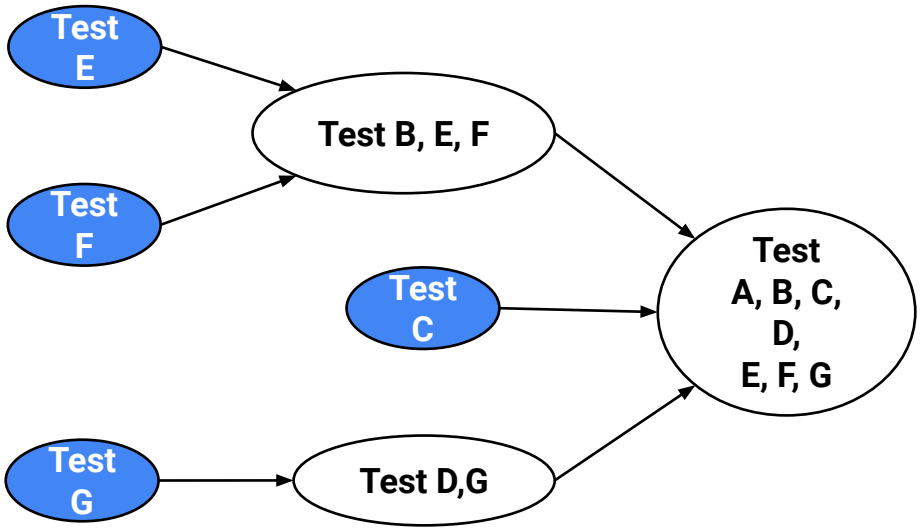


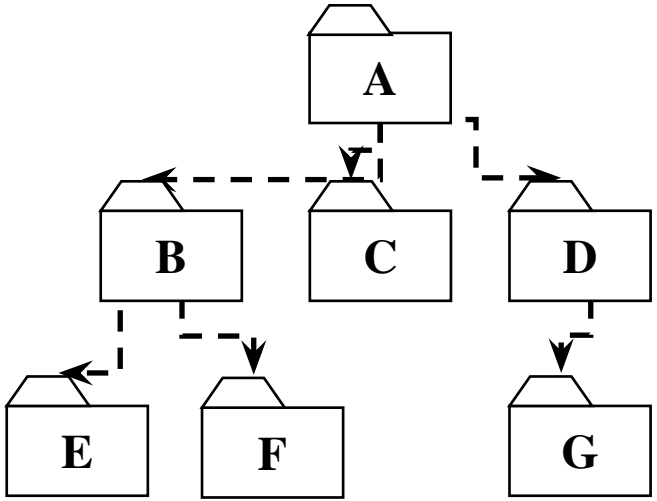


Integration Testing: Decomposition Based

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.

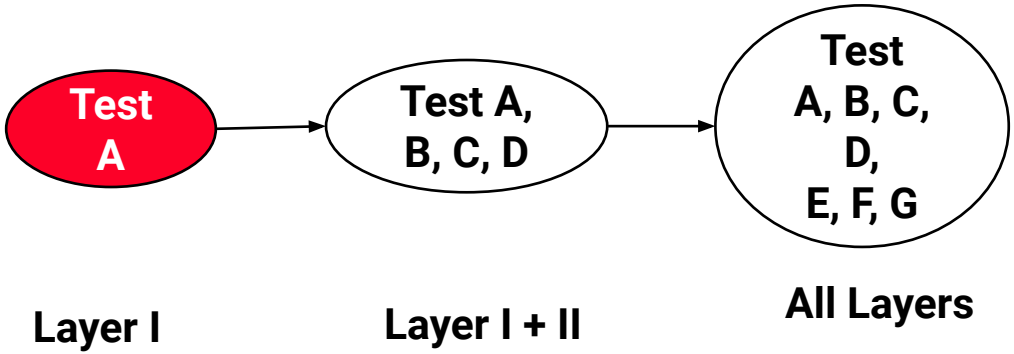


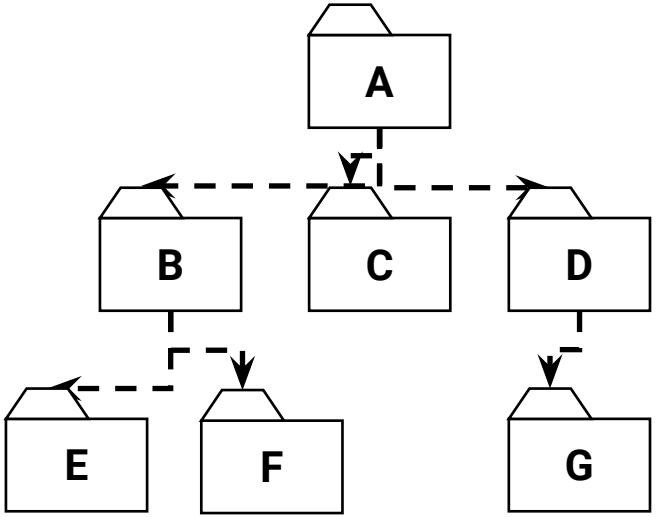


Integration Testing: Decomposition Based

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.





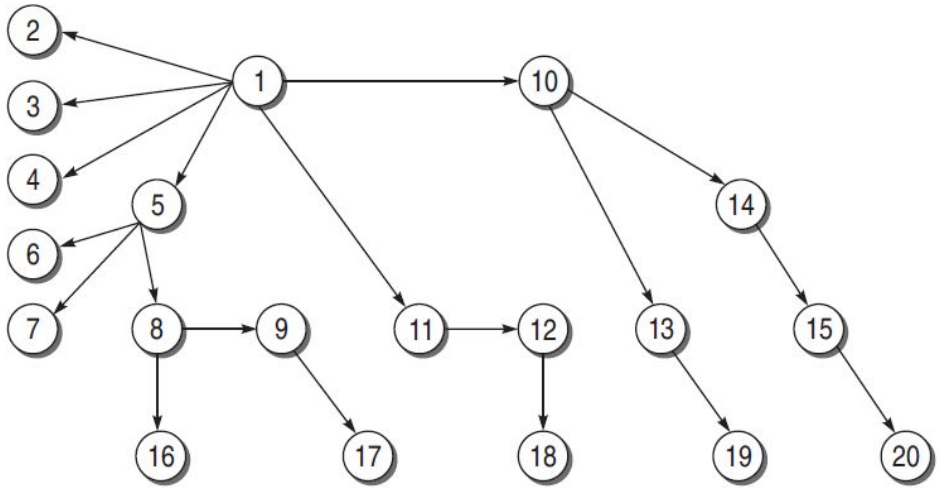
Integration Testing: Decomposition Based

Comparison between top-down and bottom-up testing

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	*(nodes – 1)* stubs are required for the subordinate modules.	*(nodes – leaves)* test drivers are required for super-ordinate modules to test the lower-level modules.

Integration Testing: Call Graph Based

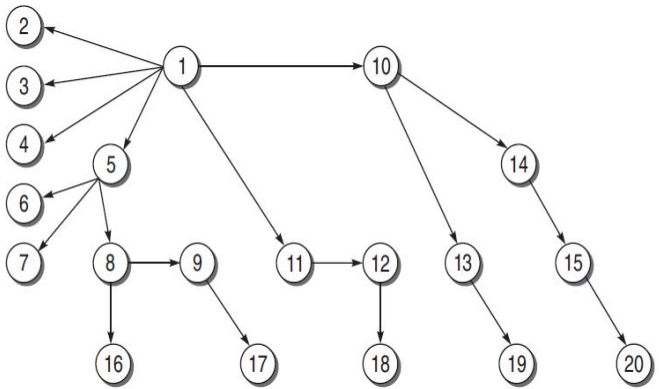
- A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- The call graph can be captured in a matrix form which is known as the adjacency matrix.
- The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers.
- If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph-based integration can be used



Call Graph

Integration Testing: Call Graph Based

Adjacency matrix

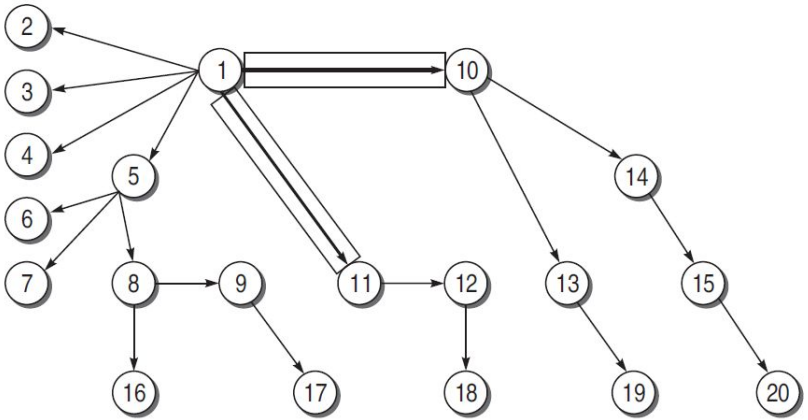


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	×	×	×					×	×									
2																			
3																			
4																			
5					×	×	×												
6																			
7																			
8								×							×				
9																×			
10												×	×						
11											×								
12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

Integration Testing: Call Graph Based

There are two types of integration testing based on call graph:

- Pairwise Integration
- Neighborhood Integration



Pairwise Integration

A form of integration testing that targets pairs of components that work together, as shown in a call graph.

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., 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.

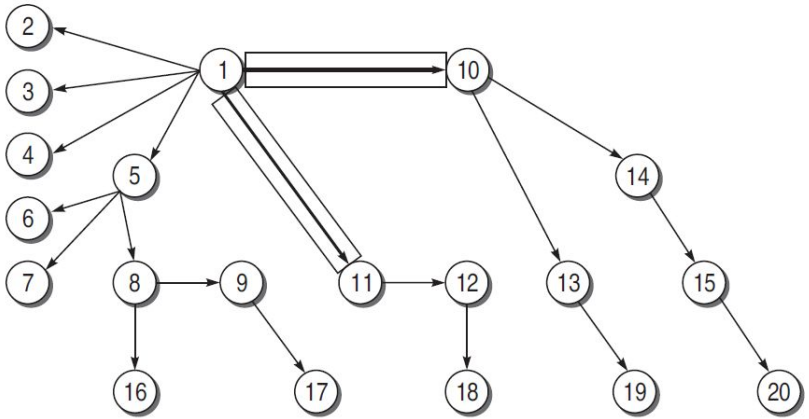
Integration Testing: Call Graph Based

Neighborhood Integration

There is not much reduction in the total number of test sessions in pairwise integration as compared to decomposition-based integration. If we consider the neighborhoods of a node in the call graph, then the number of test sessions may reduce.

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

Integration Testing: Call Graph Based



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 neighborhood integration can be calculated as:

Neighborhoods = nodes – sink nodes = 20 – 10 = 10

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

Integration Testing: Path Based

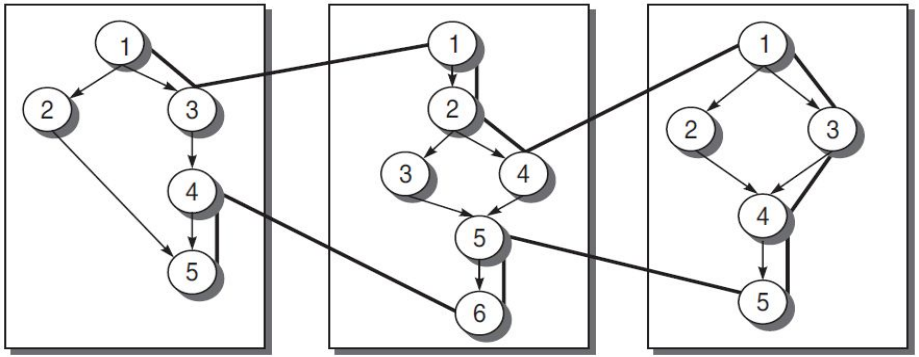
- Shift emphasis from interface testing to interactions (cofunctions) among units
- Interface-based testing is structural while interaction based testing is behavioral

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 sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.
message is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.
MM-Path It is a path consisting of MEPs and messages. it also crosses the boundary of a unit when a message is followed to call another unit.The path shows the sequence of executable statements.
MM-path graph It can be defined as an extended flow graph where nodes are MEPs and edges are messages.

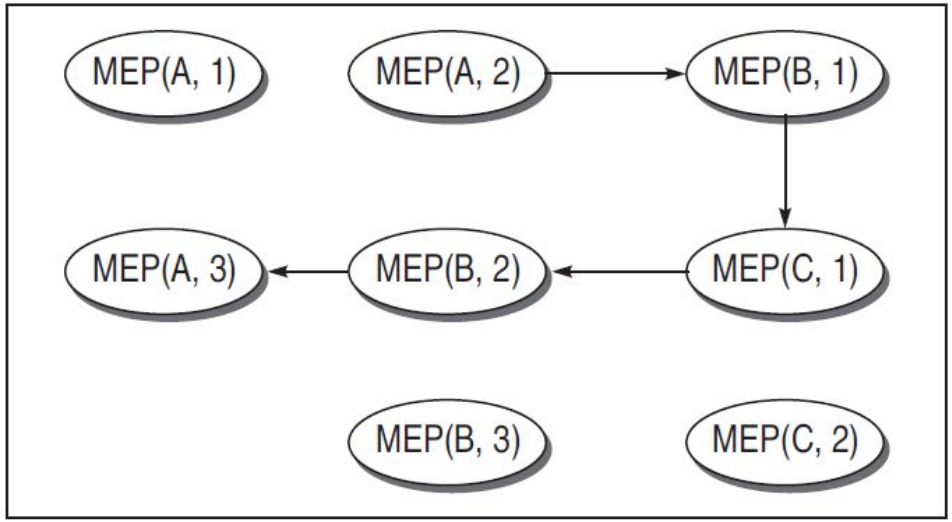
Example



	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>

Example

The MM-path graph for this example is



Mid Syllabus upto this lecture