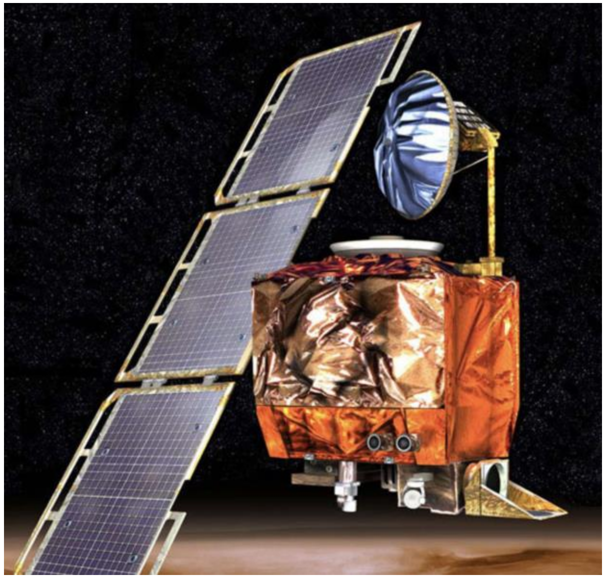
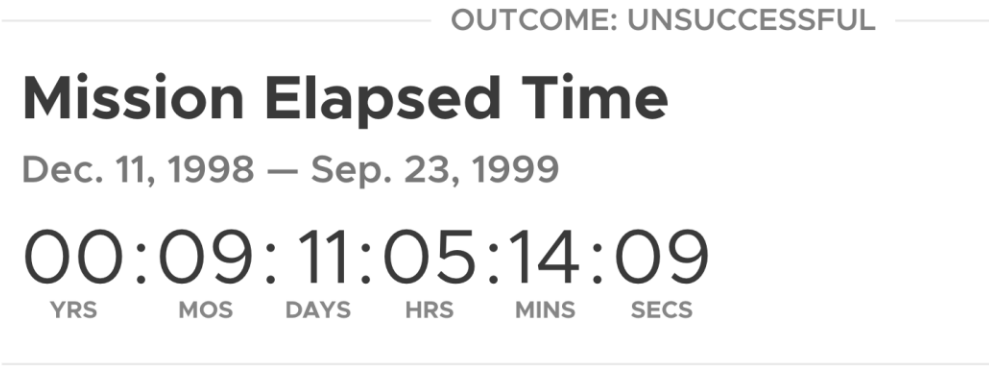
Integration Testing

Integration Testing





OUTCOME: UNSUCCESSFUL

Mission Elapsed Time

Dec. 11, 1998 — Sep. 23, 1999

00	09	11	05	14	09
YRS	MOS	DAYS	HRS	MINS	SECS

Mars Climate Orbiter 1999
travelled 416 million miles in 41 weeks; navigation error; disappeared

Unit1- Lockheed Martin Astronautics – calculations in English units (pounds)

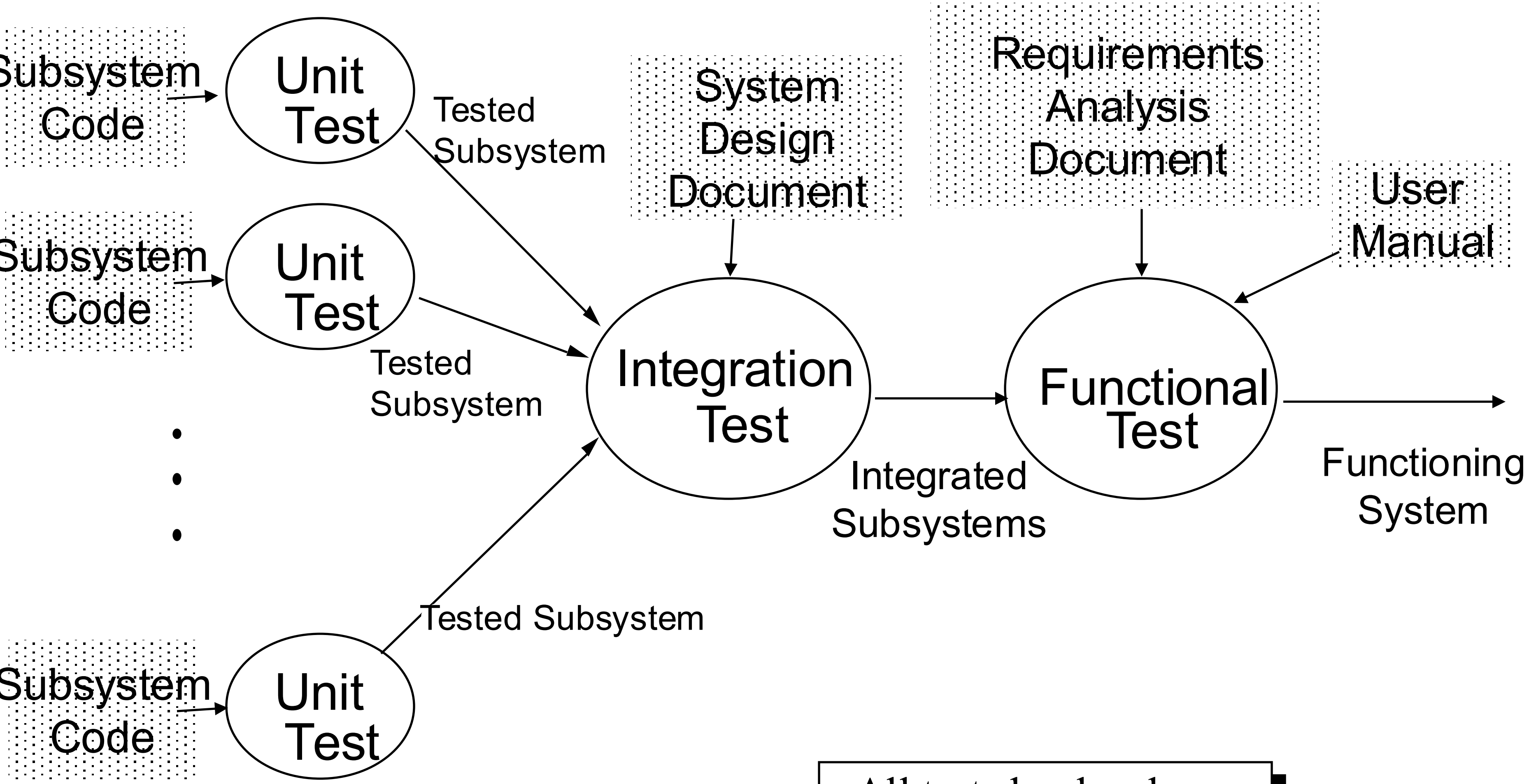
Unit-2 Jet Propulsion laboratory metric units (newtons)

Integration Testing

- integration testing - testing in which individual software modules are combined and tested as a group.

- 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.



Subsystem Code

Unit Test

Tested Subsystem

Subsystem Code

Unit Test

Tested Subsystem

Subsystem Code

Unit Test

Tested Subsystem

System Design Document

Integration Test

Integrated Subsystems

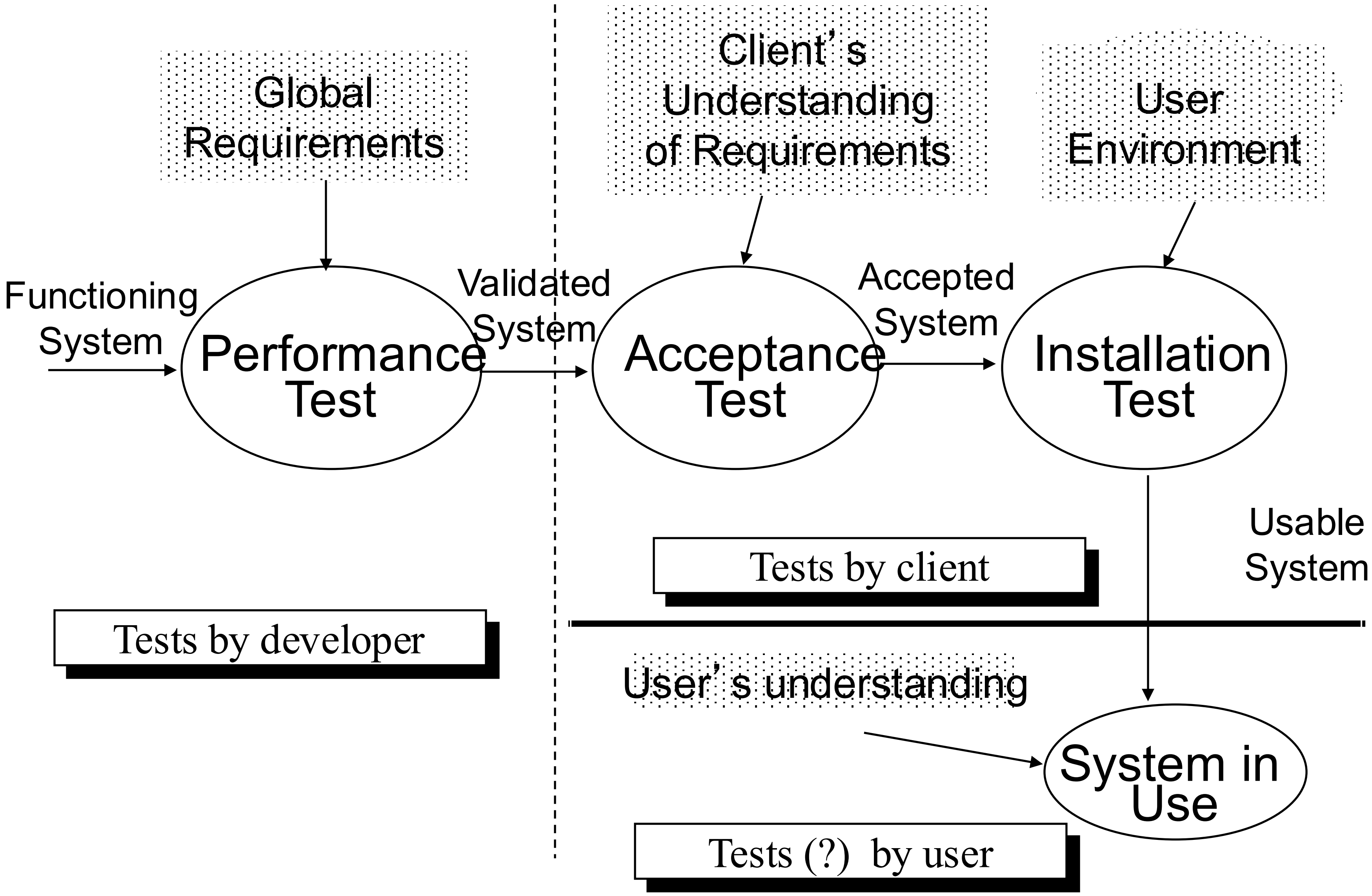
Requirements Analysis Document

User Manual

Functional Test

Functioning System

All tests by developer


Global Requirements
Client's Understanding of Requirements
User Environment
Functioning System
Performance Test
Validated System
Acceptance Test
Accepted System
Installation Test
Usable System
Tests by client
Tests by developer
User's understanding
System in Use
Tests (?) by user

Types of Testing

Unit Testing:

- □ Individual *subsystem*
- □ Carried out by developers
- □ Goal: Confirm that subsystems is correctly coded and carries out the intended functionality

Integration Testing:

- □ Groups of subsystems (collection of classes) and eventually the entire system
- □ Carried out by developers
- □ Goal: Test the ***interface*** among the subsystem
- □ Units have been separately tested

Integration and testing of software subsystems

Recall that testing uses **50-60%** of all project resources

70% of testing resources is spent on integration testing

Integration testing: main categories

Based on functional decomposition (dependency graph, dependency tree)
Structural analysis - structure of the system; *dependencies*
<u>interfaces; static</u>

Based on call graphs
(runtime) behavior; function *calls*
<u>interaction logic; dynamic</u>

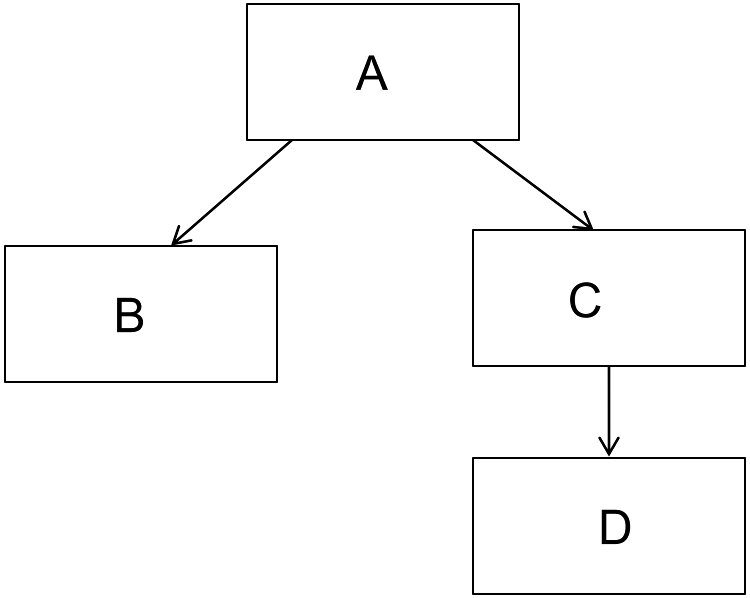
Based on execution paths
Sequences of dependencies or calls
<u>execution path; runtime (end-to-end)</u>

Dependency tree

Hierarchical representation of dependencies between modules

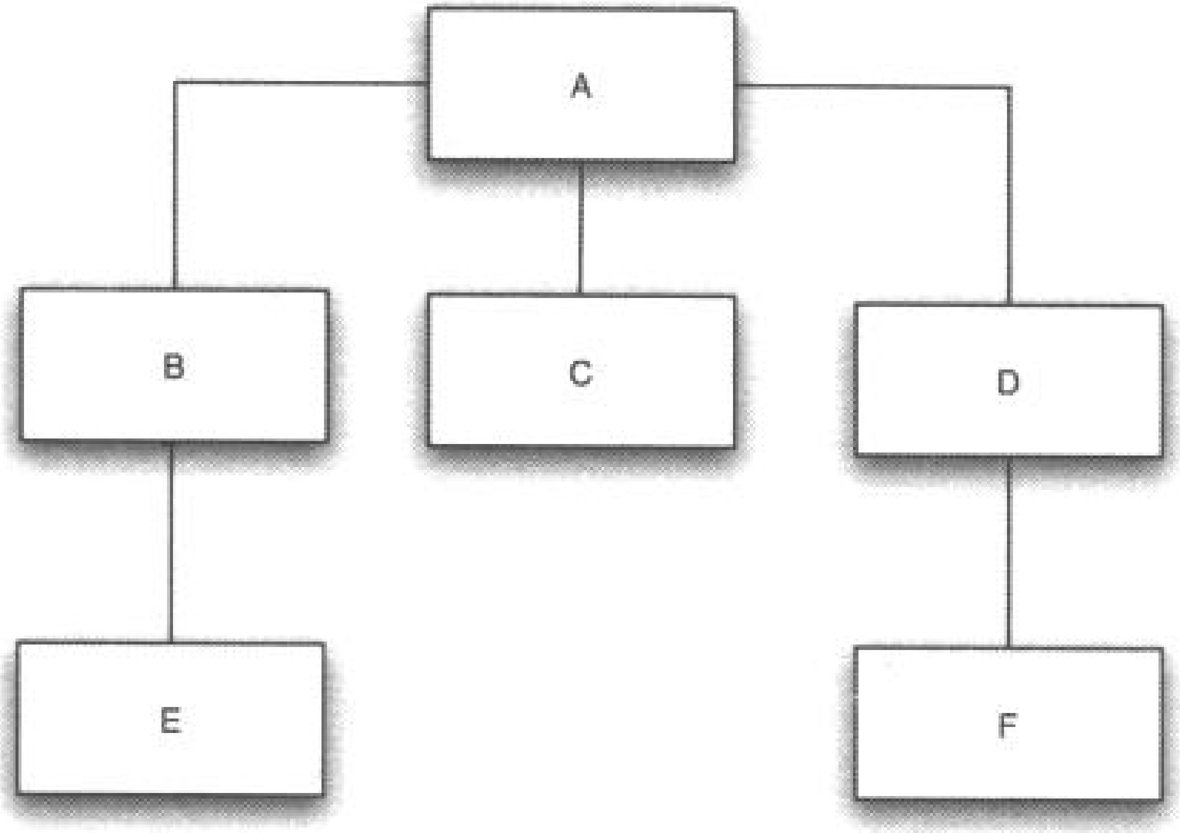
In integration testing: ***uses*** dependency

Module A uses module B == (some function in) module A calls (some function in module B)



A uses (calls) B
A uses (calls) C
C uses (calls) D

Example – A hierarchy of modules



Interaction dependencies

Function calls (usual case)

Remote procedure calls

Communication through global data

Client-server architecture

Inheritance

Calls to an application programming interface (API)

Pointers to objects

…

Decomposition-based integration testing: example

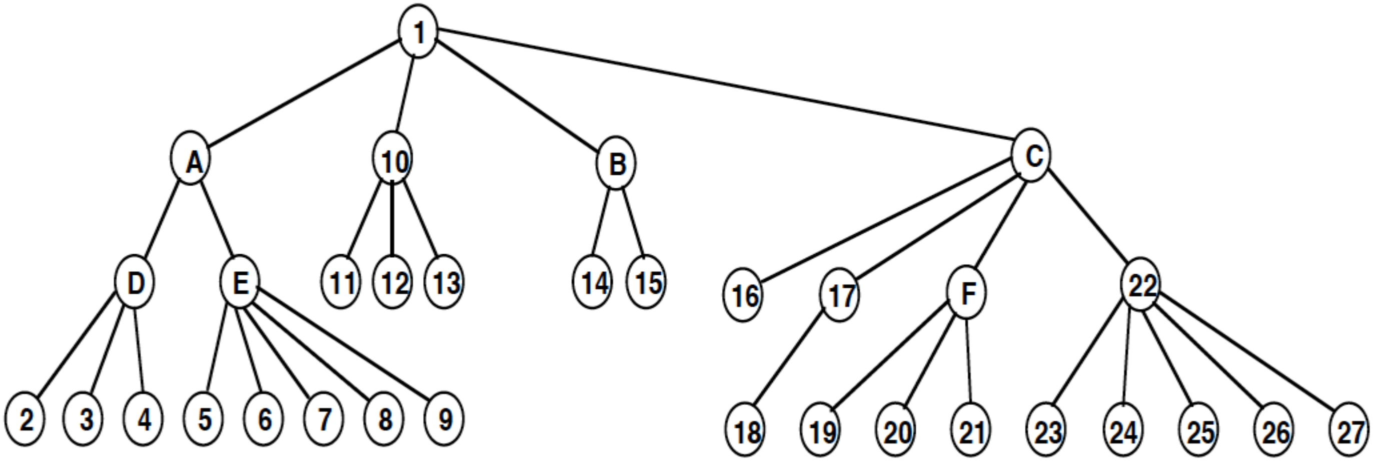
.

Unit	Level	Name
1	1	SATM system
A	1.1	Device sense & control
D	1.1.1	Door sense & control
2	1.1.1.1	Get door status
3	1.1.1.2	Control door
4	1.1.1.3	Dispense cash
E	1.1.2	Slot sense & control
5	1.1.2.1	Watch card slot
6	1.1.2.2	Get deposit slot status
7	1.1.2.3	Control card Roller
8	1.1.2.4	Control Envelope Roller
9	1.1.2.5	Read card strip
10	1.2	Central bank comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get account status
13	1.2.3	Post daily transactions
B	1.3	Terminal sense & control

Unit	Level	Name
14	1.3.1	Screen door
15	1.3.2	Key sensor
C	1.4	Manage session
16	1.4.1	Validate card
17	1.4.2	Validate PIN
18	1.4.2.1	Get PIN
F	1.4.3	Close session
19	1.4.3.1	New transaction request
20	1.4.3.2	Print receipt
21	1.4.3.3	Post transaction local
22	1.4.4	Manage transaction
23	1.4.4.1	Get transaction type
24	1.4.4.2	Get account type
25	1.4.4.3	Report balance
26	1.4.4.4	Process deposit
27	1.4.4.5	Process withdrawal

Decomposition-based integration testing: example

Unit	Level	Name
1	1	SATM system
A	1.1	Device sense & control
D	1.1.1	Door sense & control
2	1.1.1.1	Get door status
3	1.1.1.2	Control door
4	1.1.1.3	Dispense cash
E	1.1.2	Slot sense & control
5	1.1.2.1	Watch card slot
6	1.1.2.2	Get deposit slot status
7	1.1.2.3	Control card Roller
8	1.1.2.4	Control Envelope Roller
9	1.1.2.5	Read card strip
10	1.2	Central bank comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get account status
13	1.2.3	Post daily transactions
B	1.3	Terminal sense & control
14	1.3.1	Screen door
15	1.3.2	Key sensor
C	1.4	Manage session
16	1.4.1	Validate card
17	1.4.2	Validate PIN
18	1.4.2.1	Get PIN
F	1.4.3	Close session
19	1.4.3.1	New transaction request
20	1.4.3.2	Print receipt
21	1.4.3.3	Post transaction local
22	1.4.4	Manage transaction
23	1.4.4.1	Get transaction type
24	1.4.4.2	Get account type
25	1.4.4.3	Report balance
26	1.4.4.4	Process deposit
27	1.4.4.5	Process withdrawal



Integration and testing of software subsystems

Integration
Testing
of software subsystems (modules) into a single system

Two main approaches**:**

•**Test each module independently, combine the modules [non-incremental, big bang testing / integration]**

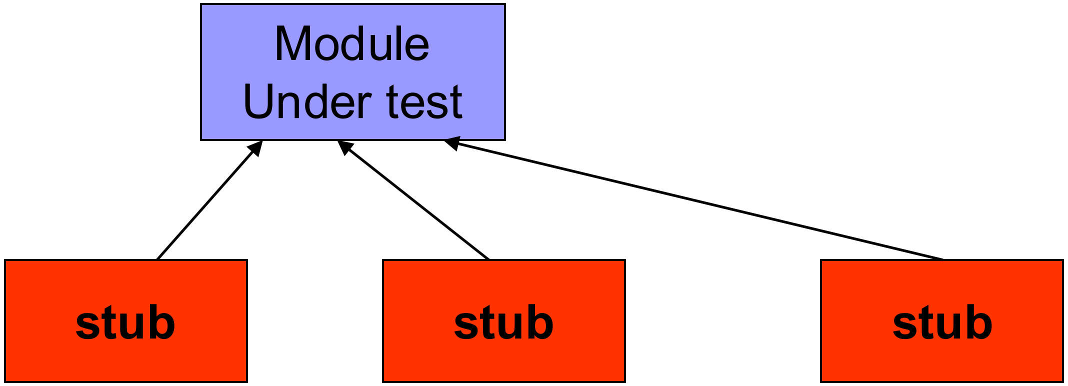
•**combine the next module to be tested with the set of the previously tested modules**
[incremental testing / integration]

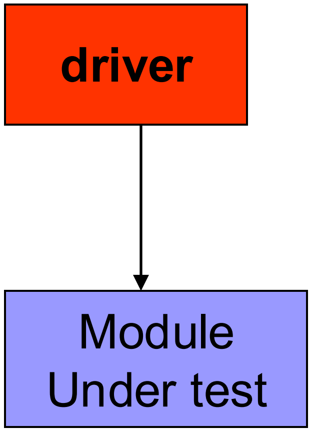
Non-incremental Integration Testing

Non-incremental testing

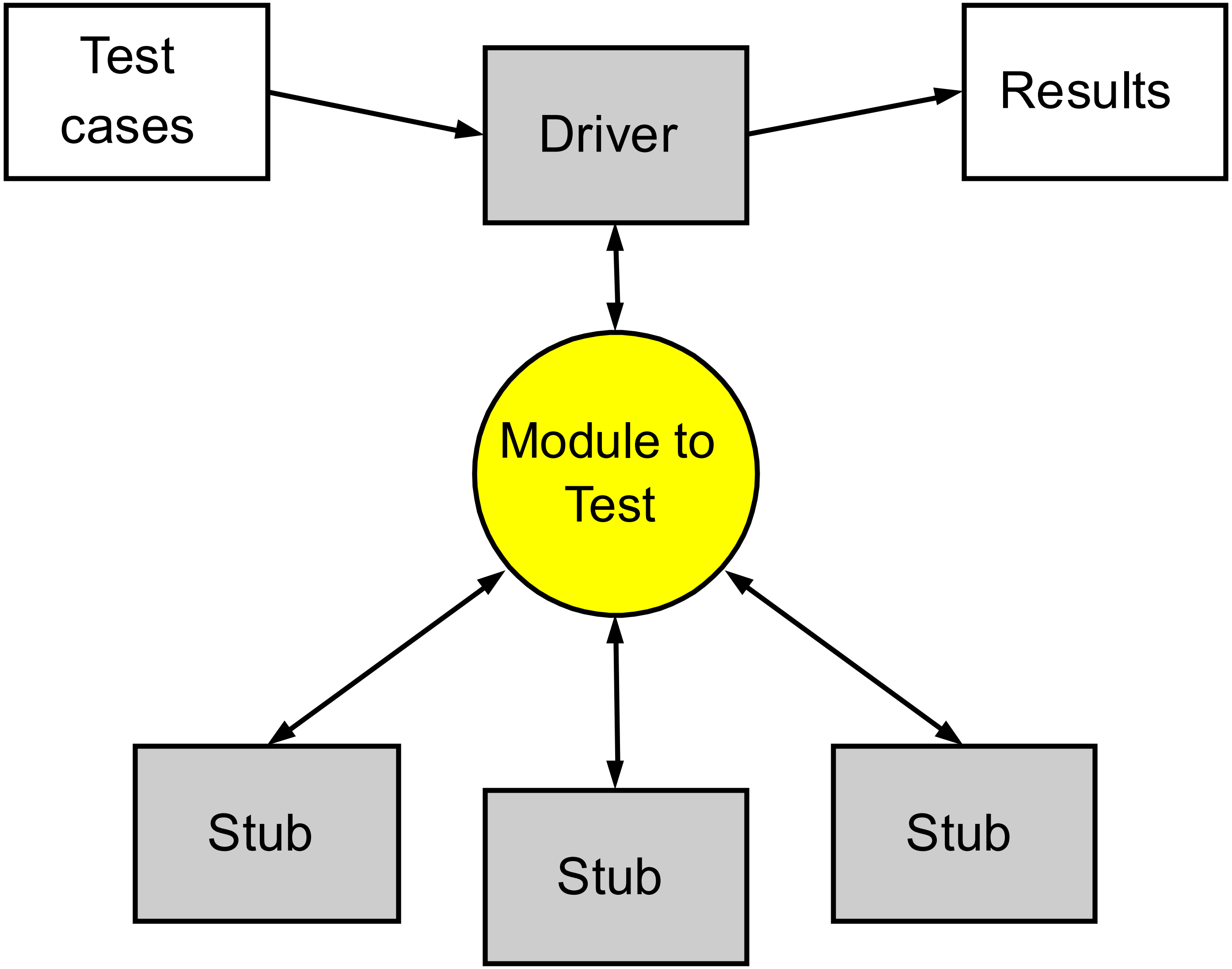
Module testing realized for each module separately (module regarded as a stand-alone entity)
Modules could be tested at the same time or in series

Testing modules requires **STUBS** OR A **DRIVER**





Scaffolding: stubs and drivers



Stubs

- Do nothing
- Validate the inputs
- Send a message to a log
- Return a hard-coded answer regardless of the input
- Select an answer from a pool of hard-coded answers
  - Cycle through the pool or randomly select one
- Randomly generate an answer
- Prompt the user for the answer
- Simple implementation of the module that is slower, less accurate, or somehow less capable than the real module
- Pause for awhile to simulate the time taken by the real module

Drivers

- Invokes the module with fixed inputs
- Compares actual outputs with expected outputs
- Records failure if expected and actual outputs do not match
- Normally continues to execute even if a test case fails

- Drivers and stubs must be designed together

- Since stubs do not produce “real” outputs, the expected results in the driver must take into account the “fake” behavior of the stubs

Non-incremental (Big Bang) Testing

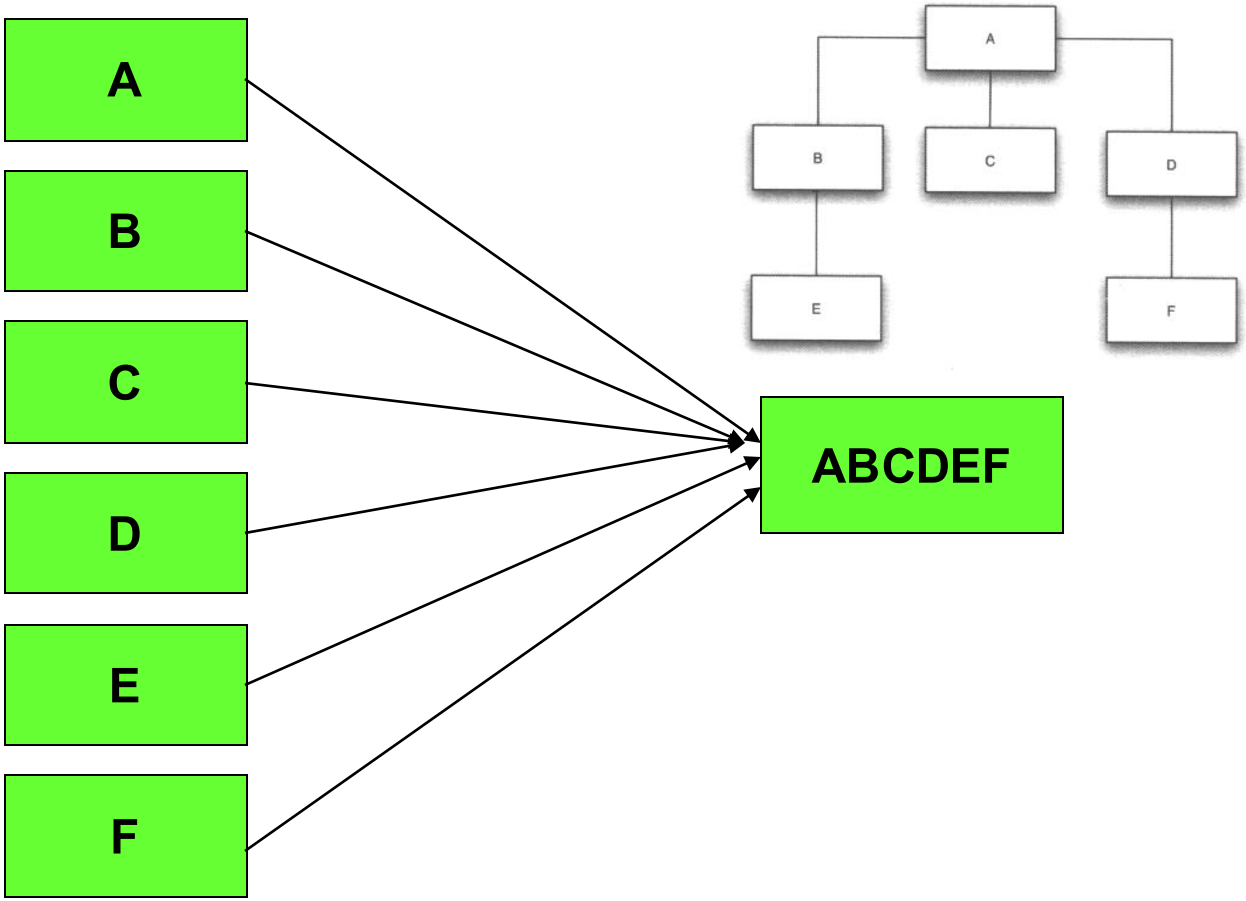
In Big Bang Integration testing, individual modules of the programs not integrated until **everything** is ready.

'Run it and see' approach.

System is integrated **without any formal integration testing**, and then run to ensure that all the components are working properly.

Non-incremental testing

Integration realized in a single step (big bang) and system is tested as a single entity



A

B

C

D

E

F

ABCDEF

Non-incremental testing: main features

More opportunities for parallel activities; significance in large projects

May be the only feasible approach for a monolithic system (unstructured systems, no design, components tightly coupled)

Could be suitable for small systems

Non-incremental testing: main features

Amount of work could be large: number of stubs and drivers

Possible mismatching interfaces: modules do not “see one another” until the end of the process

Debugging could be difficult. It is difficult to tell whether defect is
In the component or interface

Approach is ambiguous and opportunistic

When failure (not *if*, but *when*) is observed, isolation of the fault is difficult

Bottom-up Integration Testing

Bottom-up integration testing

Strategy starts with the terminal modules (that do not call other modules)

We need drivers

There is no concept of a skeleton of the system (integration is late and time to working system is longer).

Well-suited for components with robust and stable interface

Bottom-up design methodology was used

Can provide an early assessment of a unit that must implement a critical requirement

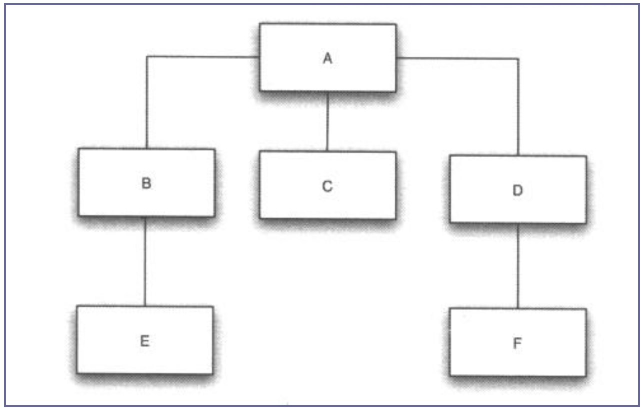
Work may proceed in parallel

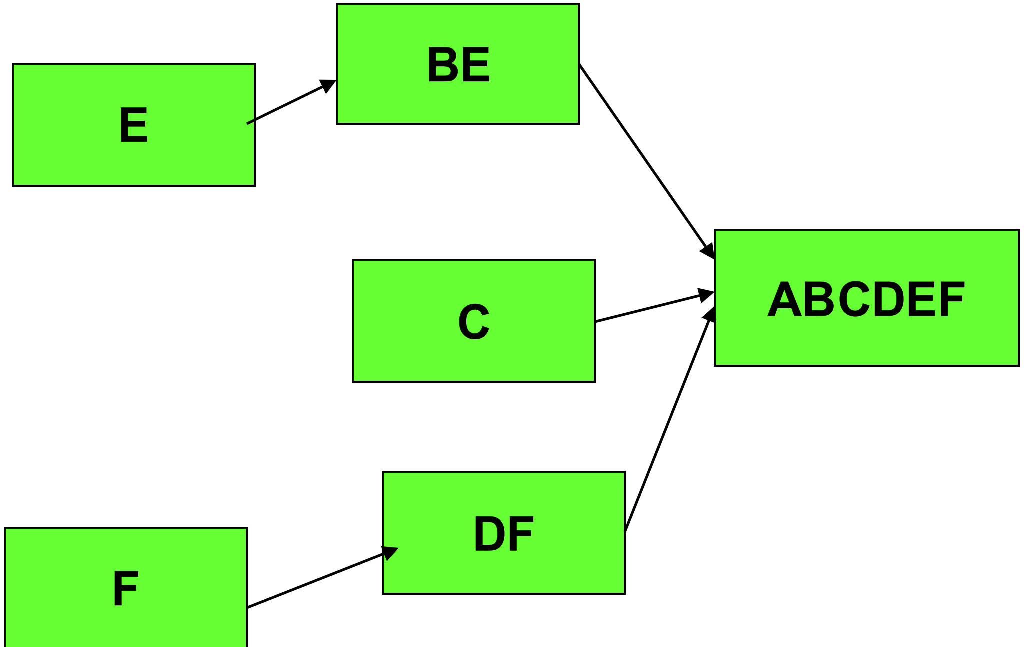
Bottom up integration testing

Bottom-up: start from the lower end and move forward

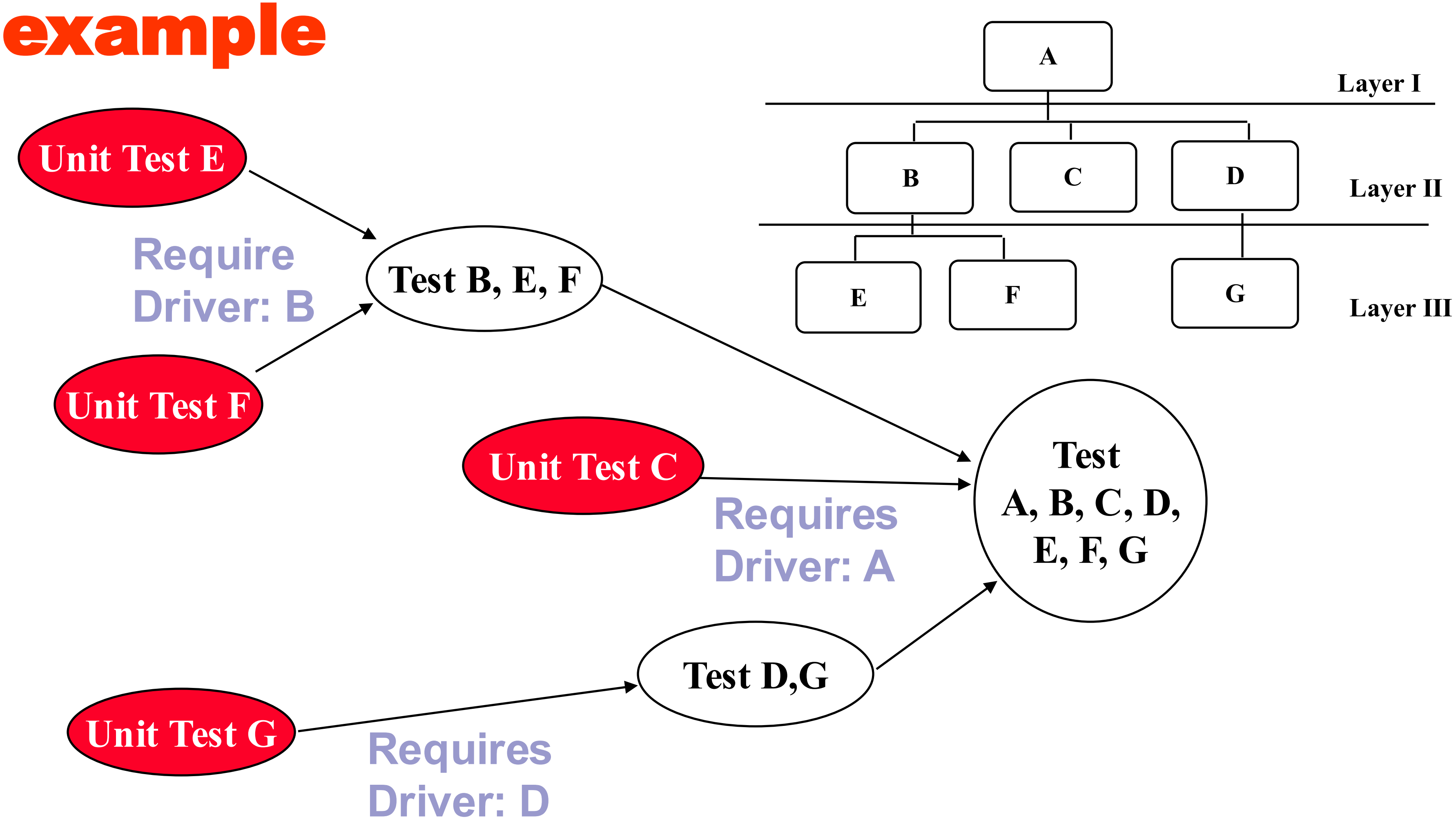
E, C, F – we need drivers here (3)

BE and DF (B and D are not tested in isolation but with E and F)
we need 2 drivers





Bottom up integration testing-example



Bottom-up Integration Testing

As and when code for other module gets ready, these drivers are replaced with the actual module.

In this approach, lower level modules are tested extensively thus make sure that highest used module is tested properly.

Top-down Integration Testing

Testing strategy

Test the top layer (or the controlling subsystem) first

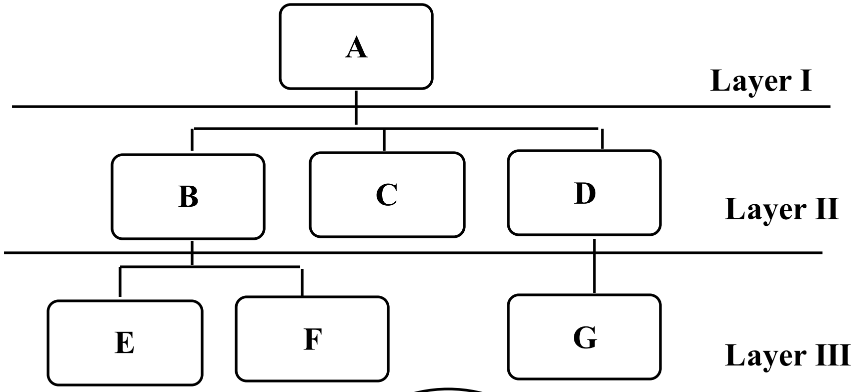
Combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems

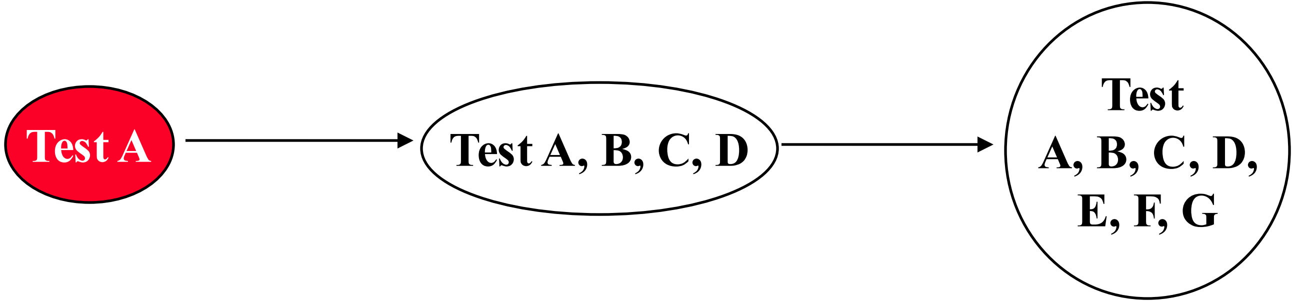
Do this until all subsystems are incorporated into the test

Lower level modules are normally simulated by stubs which mimic functionality of lower level modules.

As lower level code has been added, stubs are replaced with the actual components.

Example





Test A

Test A, B, C, D

Test
A, B, C, D,
E, F, G

Layer I

Layer I + II

All Layers

Requires stubs: B C D

E F G

Top-down integration testing

Start from the top (initial module)

No single procedure to select next module to be incrementally tested

Just choose one for which one of the calling modules must have been tested previously

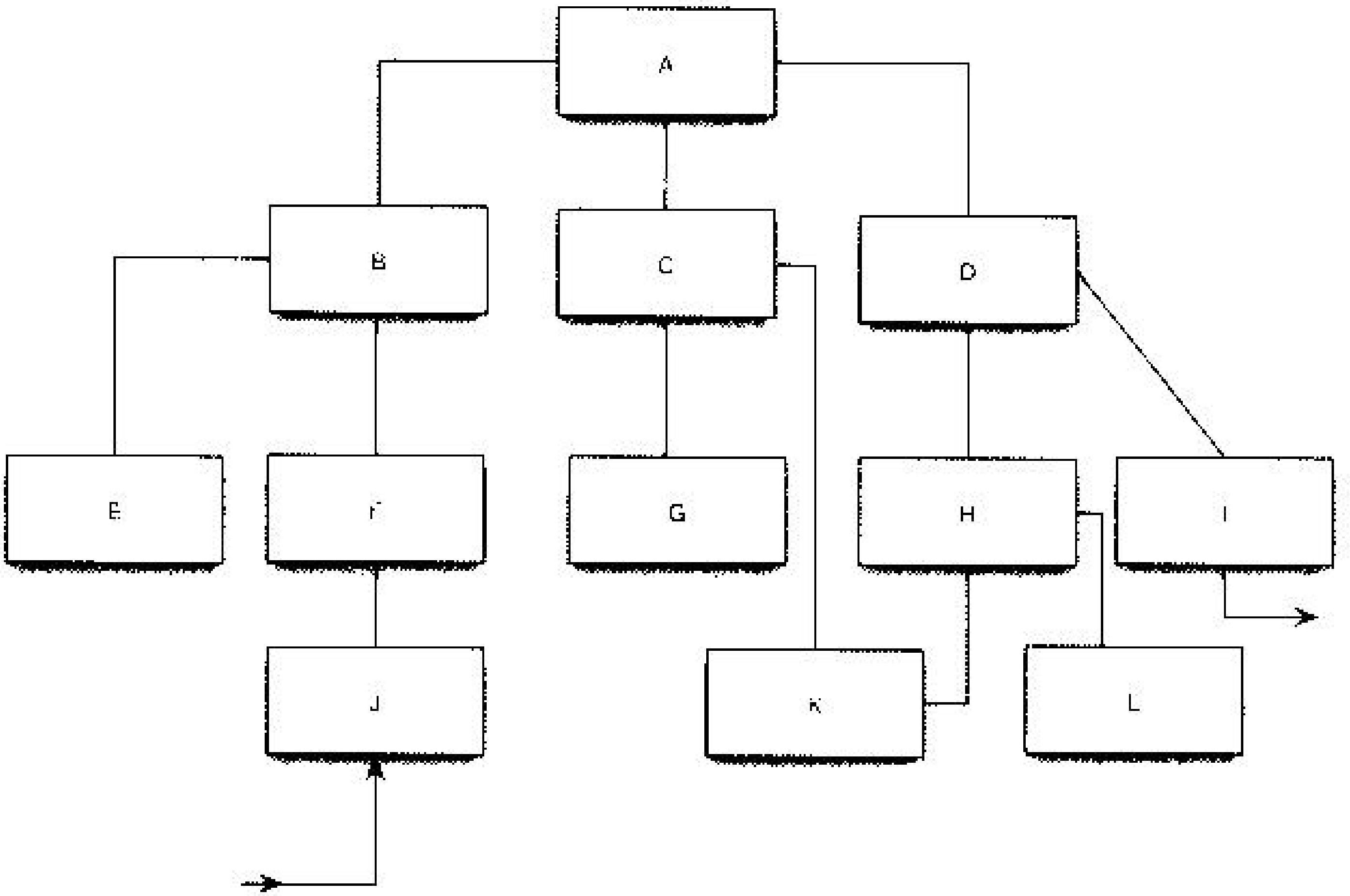
Stubs are needed for modules that are called

Avoidance of commitment to an unstable interface (lower level interfaces undefined or likely to change)

Constructing stubs is not trivial – sometimes it is misunderstood (say dummy stubs or " we got so far")

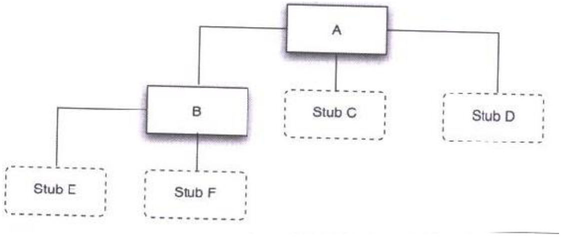
Stubs feed data to modules at the higher level; we may require multiple versions of the same stub

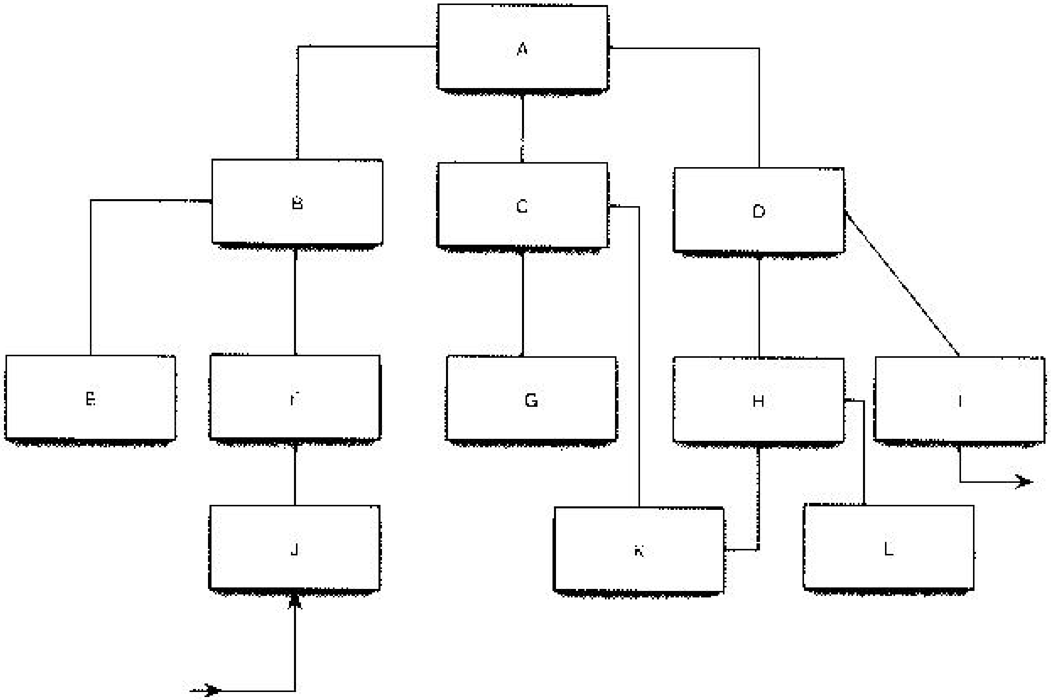
Top-down testing



Module J: I/O operations **Module I: write operations**

Top-down testing





Possible sequences of modules:

A B C D E F G H I J K L

A B E F J C G K D H L I

A D H I K L C G B F J E

A B F J D I E C G K H L

General strategies:

Depth –first
Breadth – first

Top-down testing: guidelines

1. RISK DRIVEN

If there are critical sections of the system, design the sequence such that these sections are added as early as possible

(critical – complex module, new algorithm, module that is suspected to have some faults, most frequently used…)

2.SCHEDULE DRIVEN

To the extent possible, integrate modules as they become available

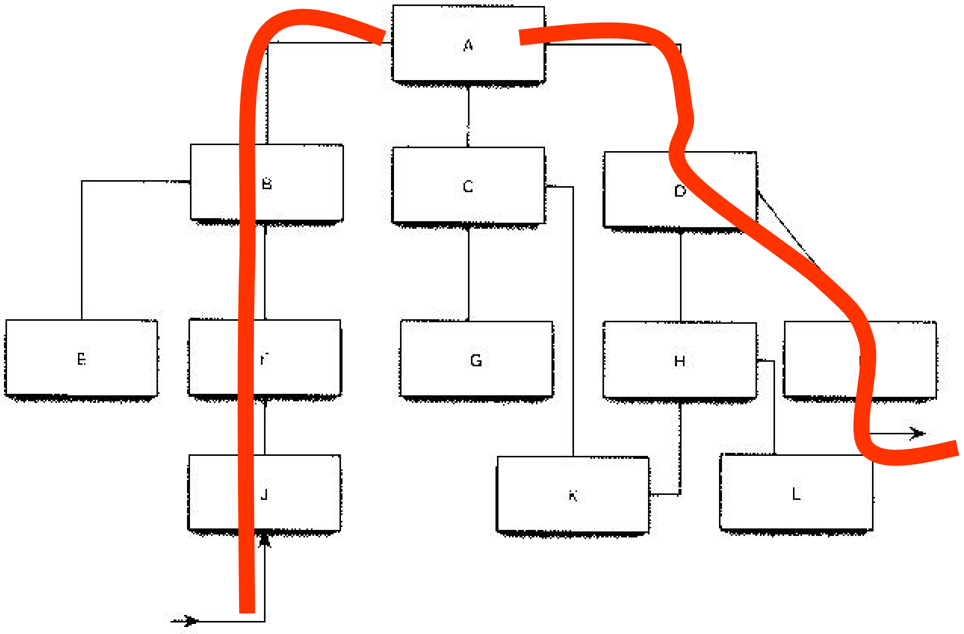
Top-down testing: guidelines

3. **FUNCTION DRIVEN**

Design the sequence such that the I/O modules are added as early as possible

For instance, as J, I are I/O modules, the sequence might be

A B F J D I C G E H K L



Top-down testing: general observations

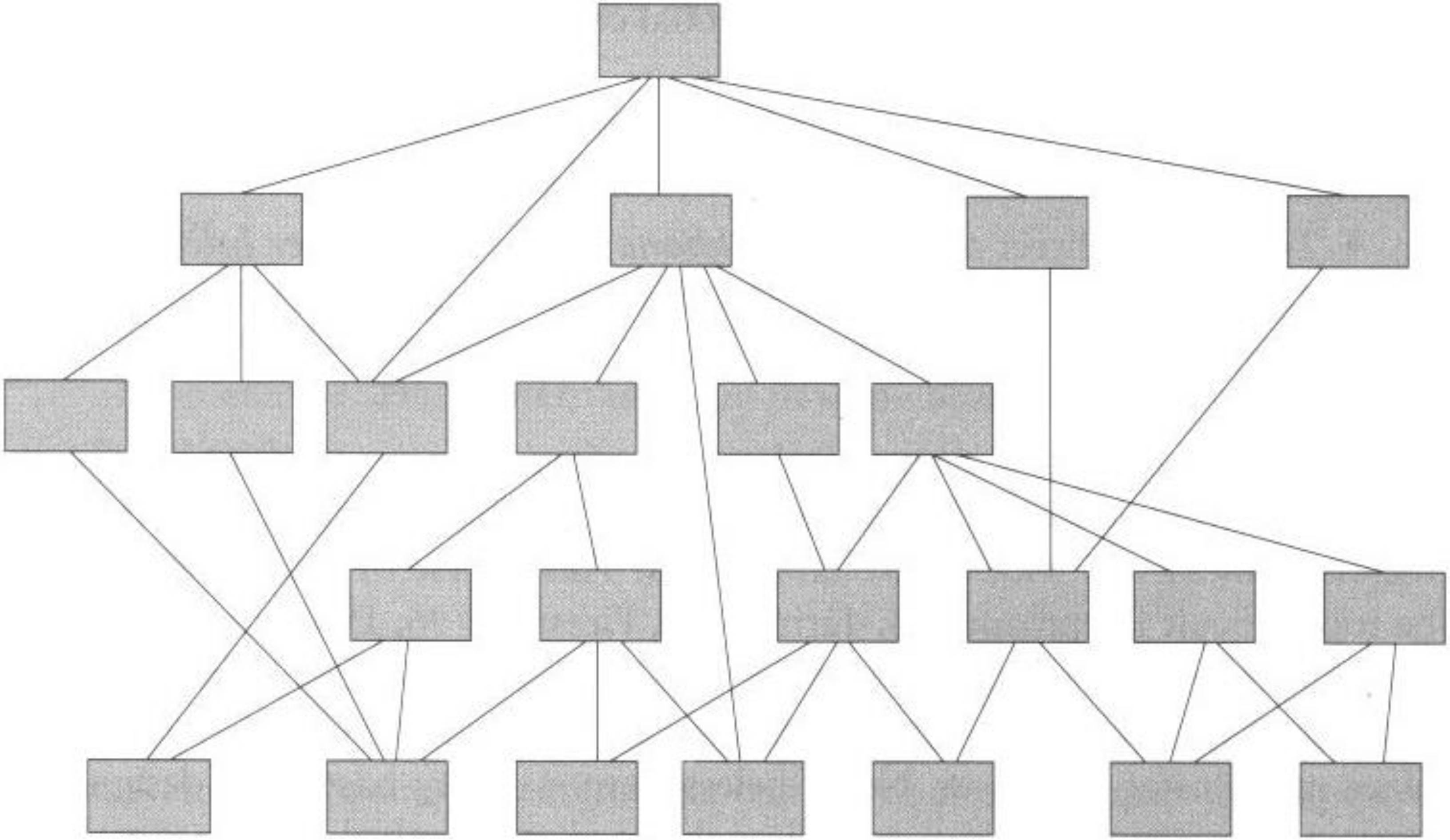
Skeleton of the entire system

Integration occurs quite early

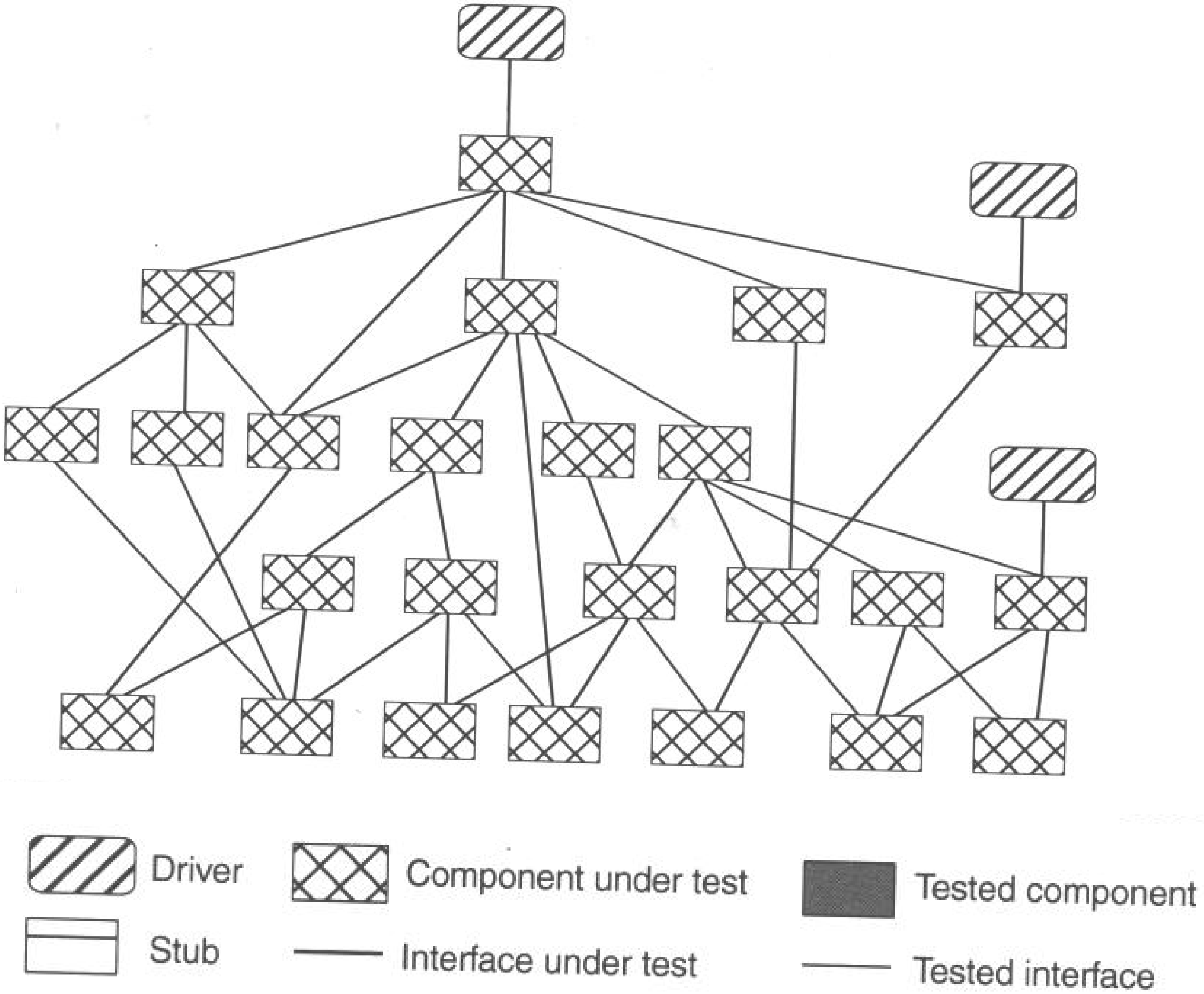
Time to working system is short

Difficulties in embedding test data in stub modules

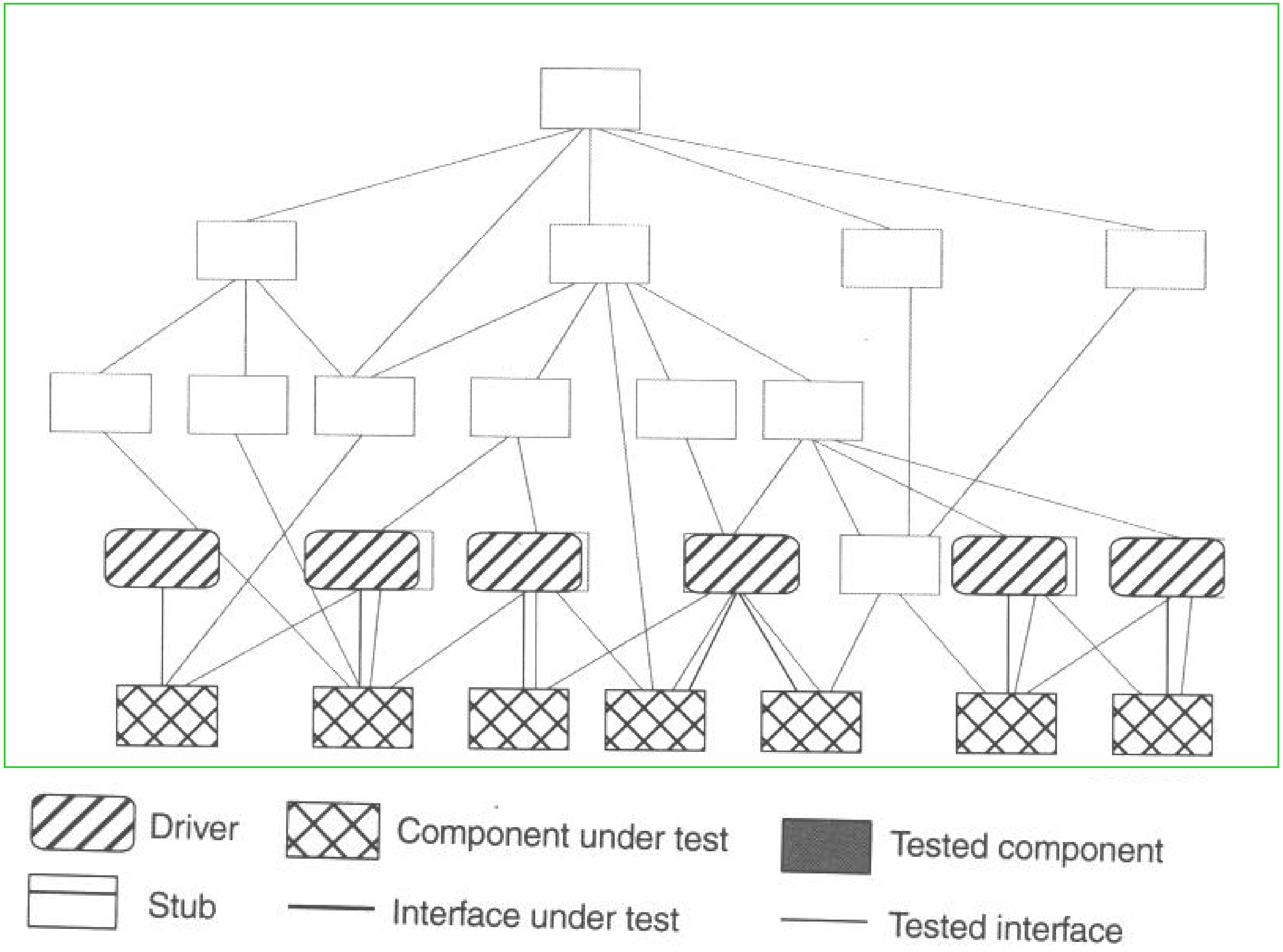
Modules and dependency tree



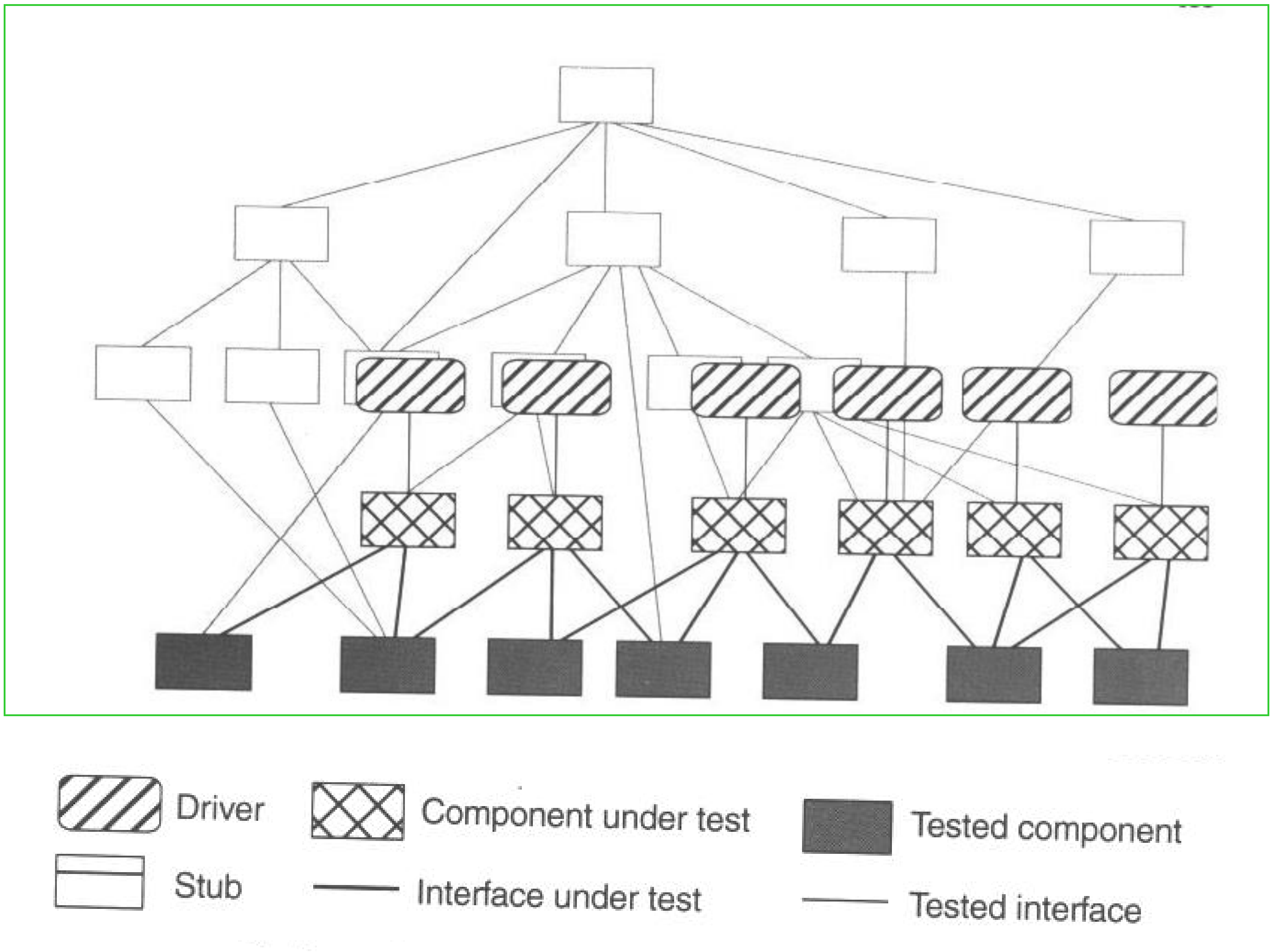
Big Bang Integration



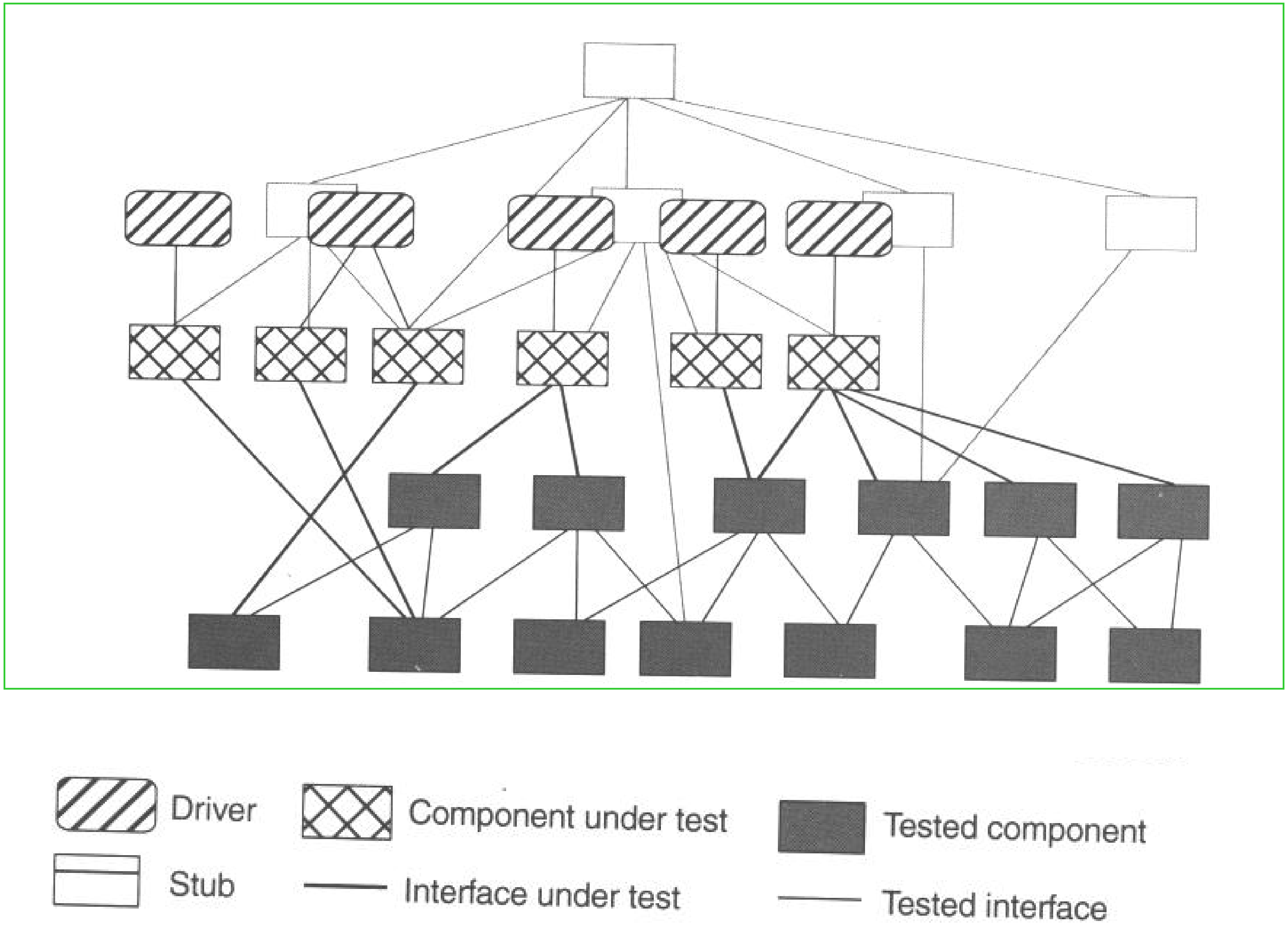
Bottom Up Integration (1)



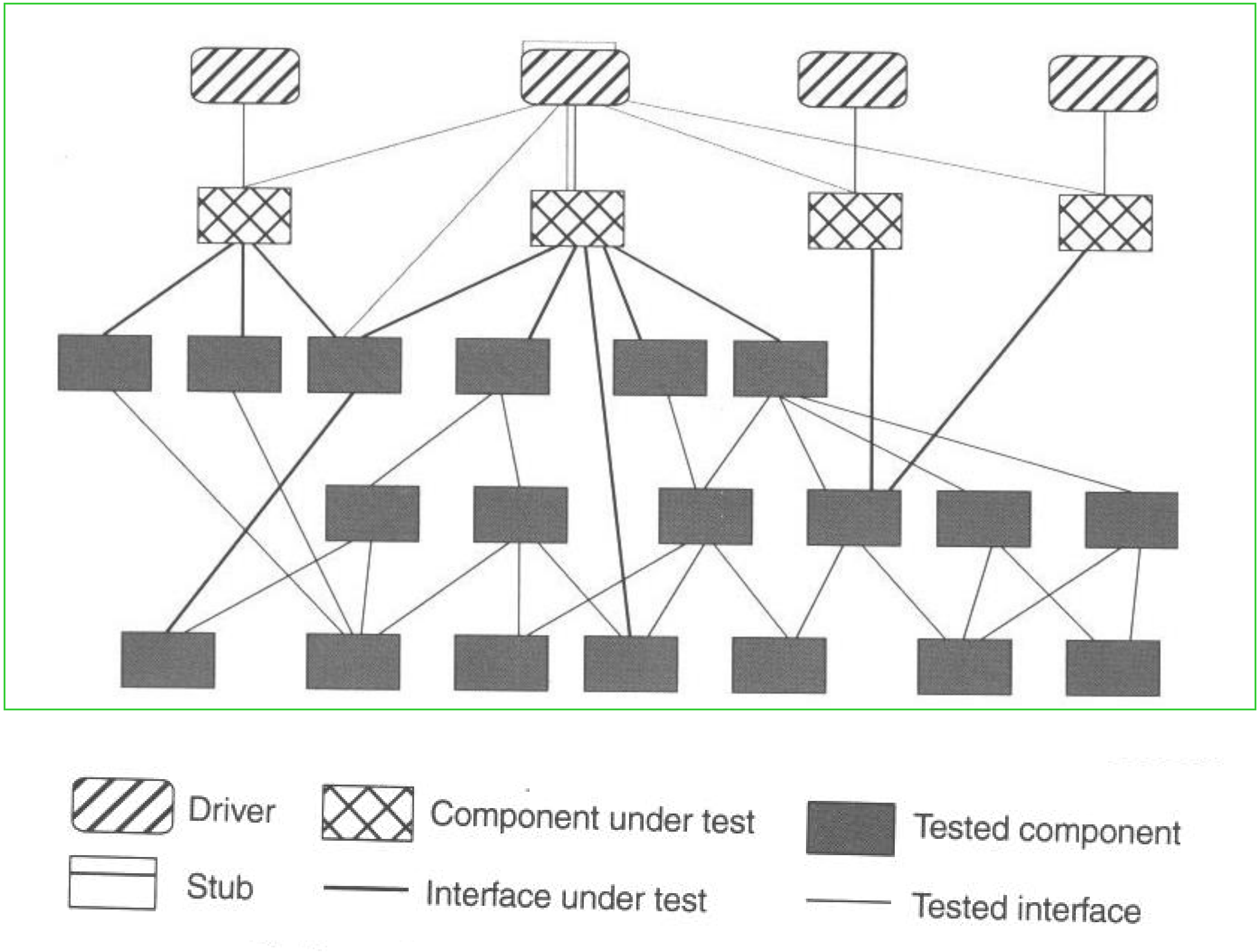
Bottom Up Integration (2)



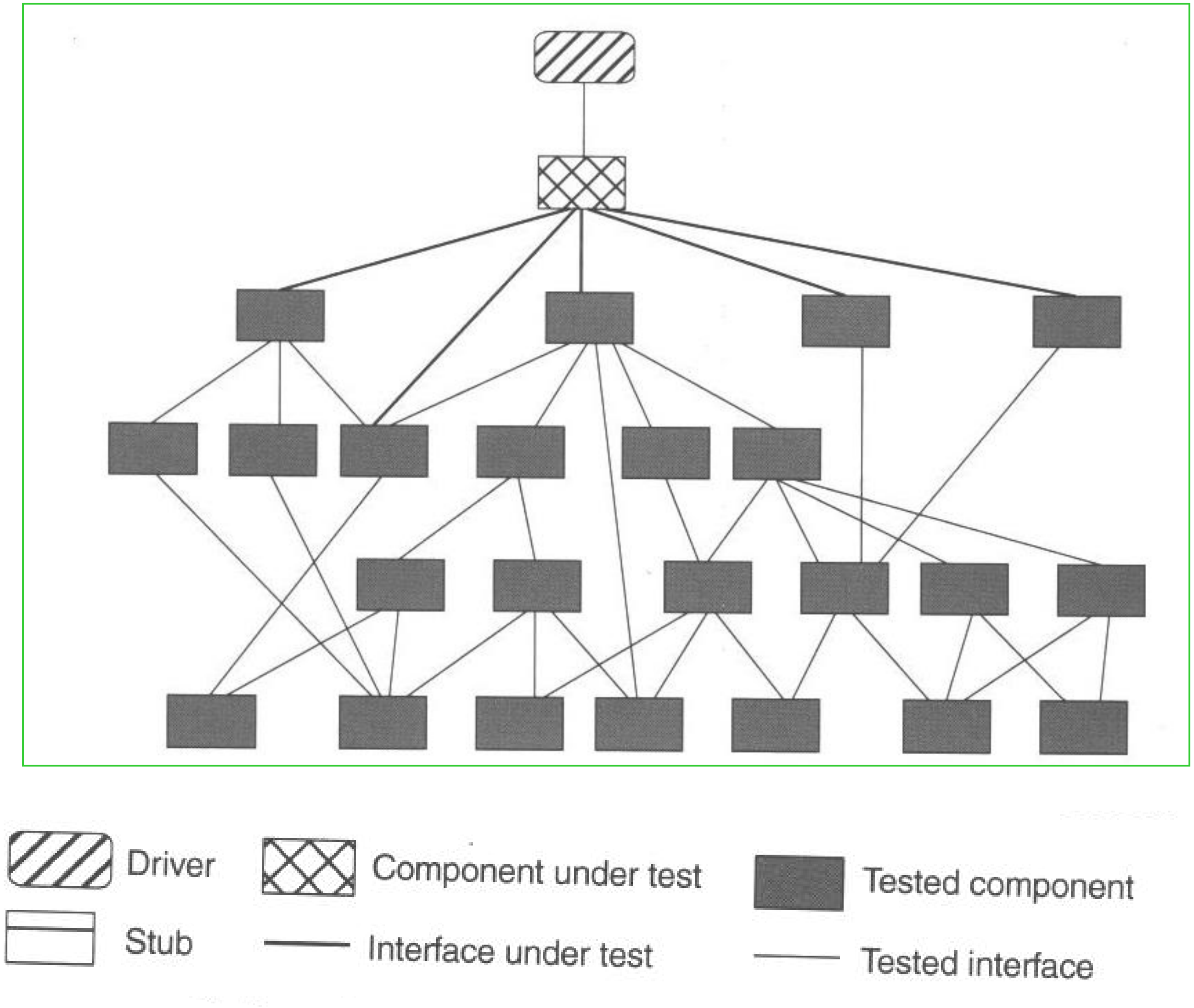
Bottom Up Integration (3)



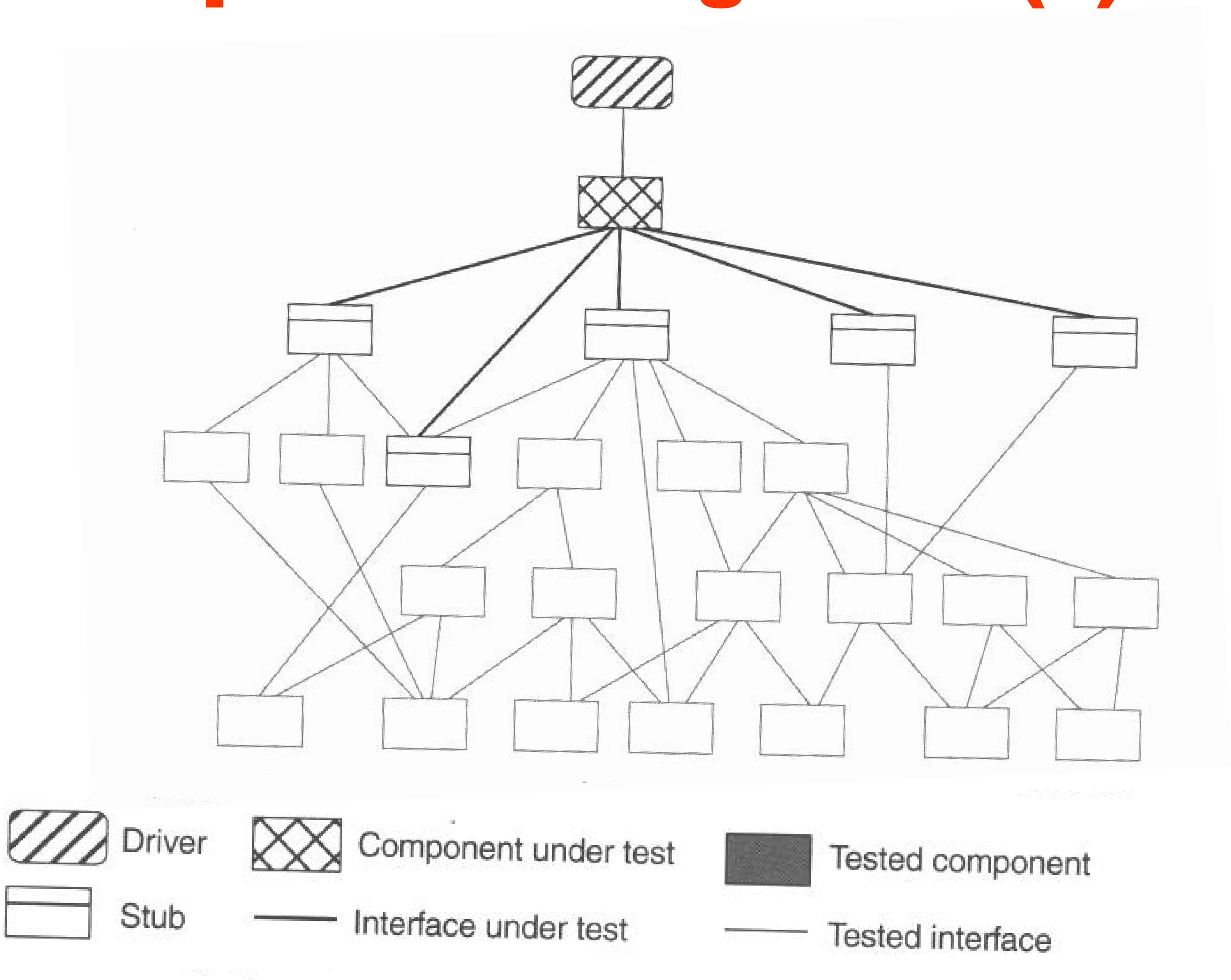
Bottom Up Integration (4)



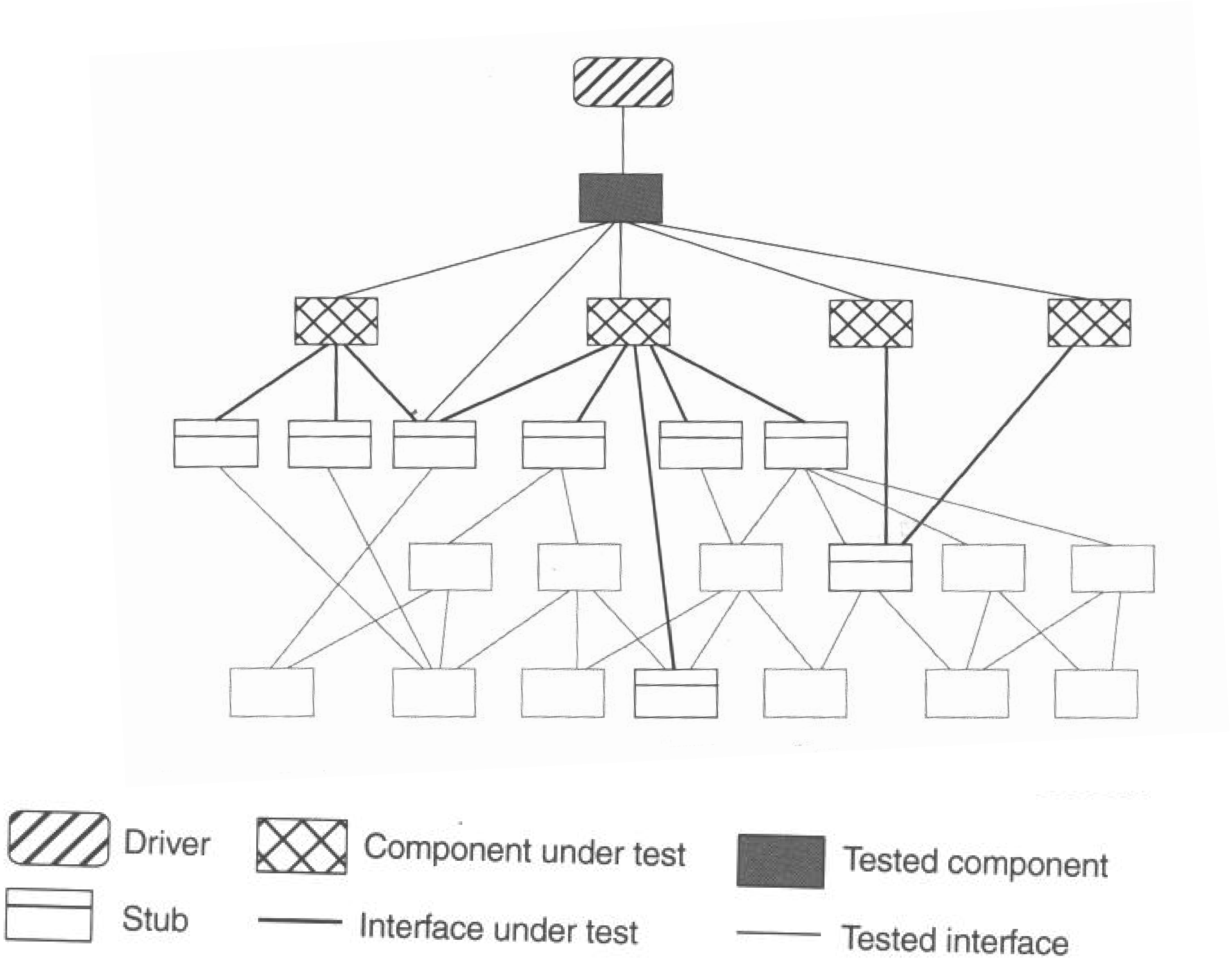
Bottom Up Integration (5)



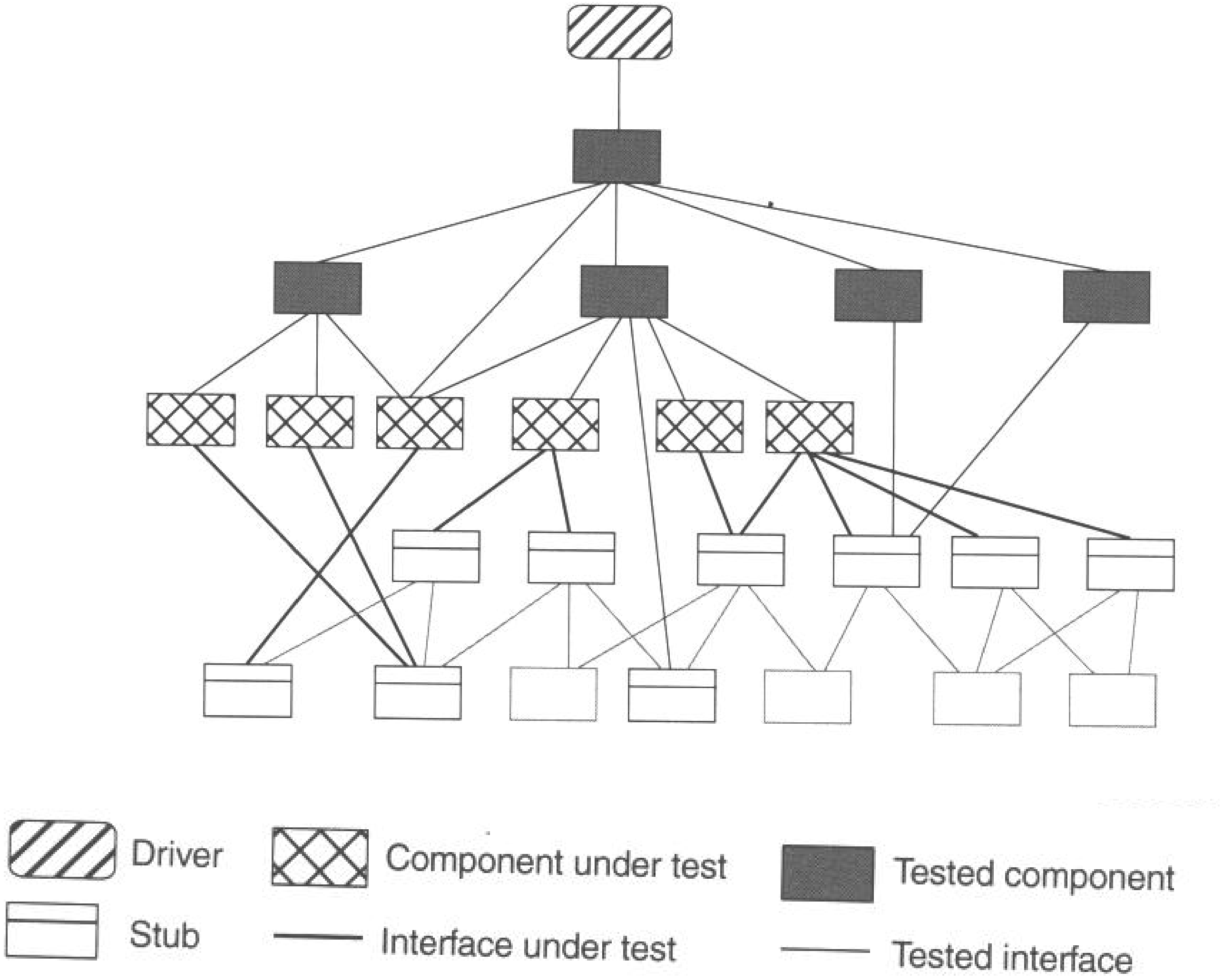
Top-Down Integration (1)



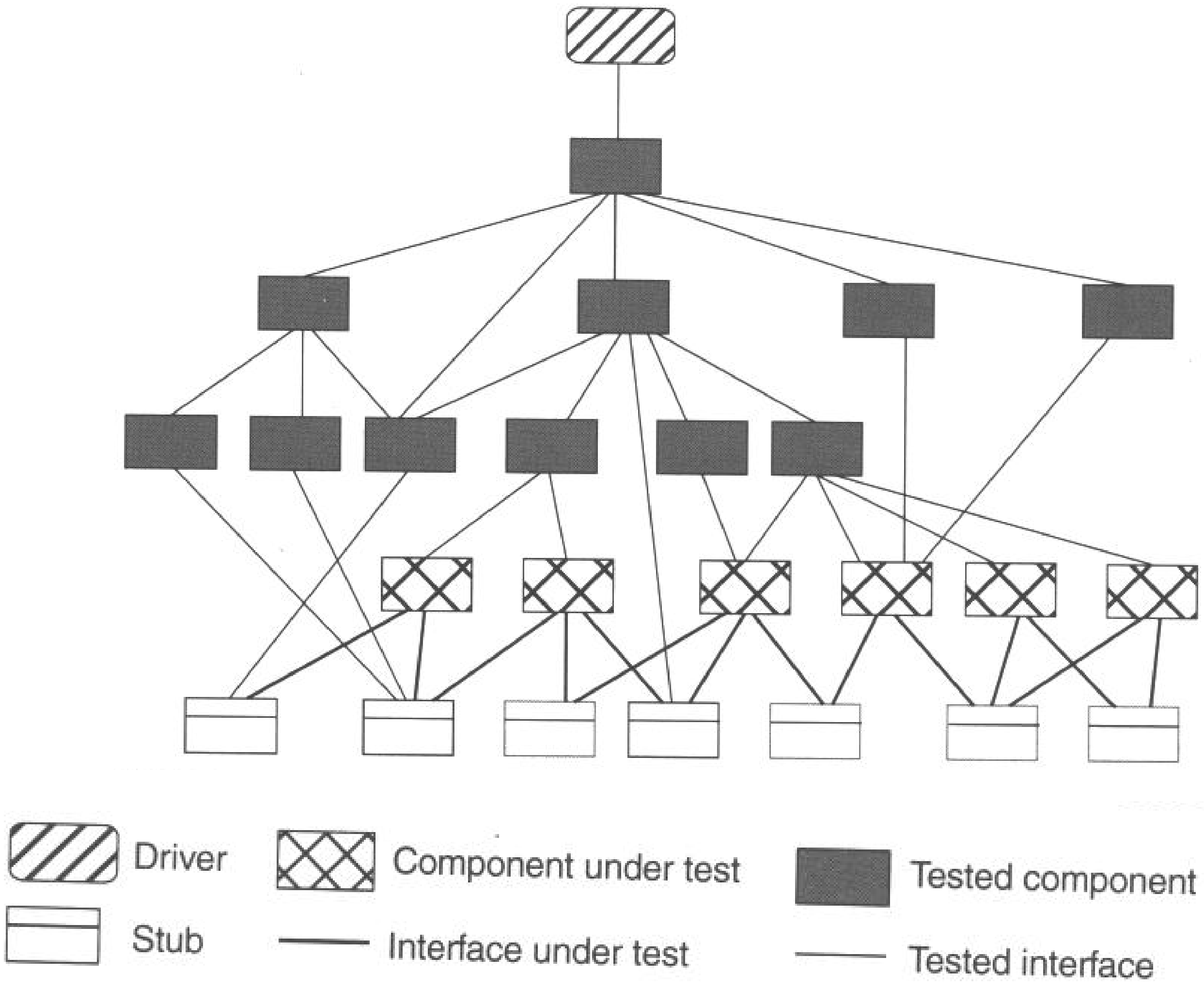
Top-Down Integration (2)



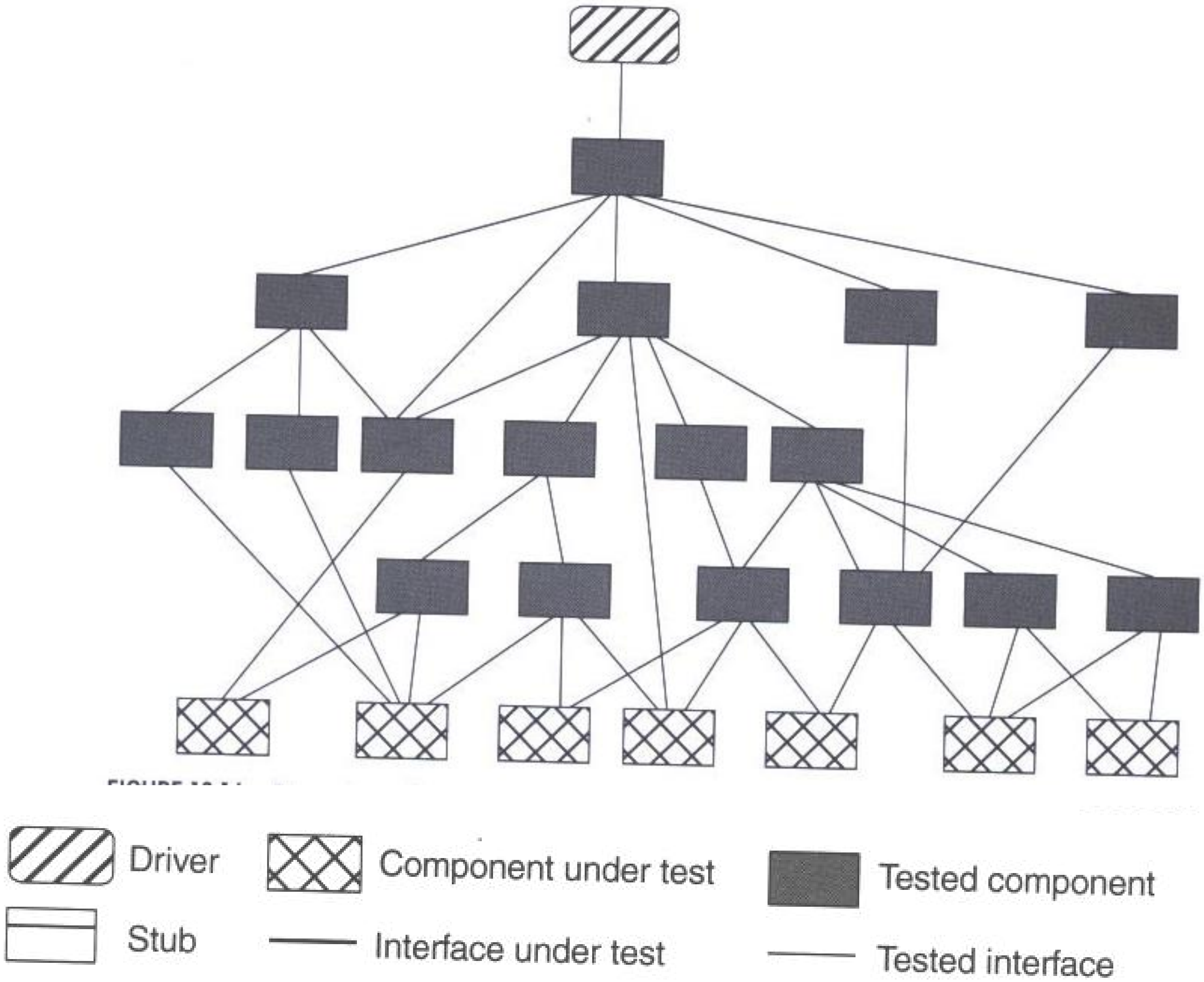
Top-Down Integration (3)



Top-Down Integration (4)



Top-Down Integration (5)



Top-down and bottom-up integration testing: a comparative overview (1)

Architecture validation

Top-down: more likely to discover faults in architecture at an early stage of the development

Bottom-up: high level design not validated until a late stage of the process

System demonstration

Top-down: limited, working system is available at an early stage

Bottom-up: late; if the system constructed from reusable components, it may be possible to offer a similar demonstration

Top-down and bottom-up integration testing: a comparative overview (2)

Test implementation

Top-down: development of stubs (simulating lower levels of system)
Bottom-up: development of drivers (simulation of components’ environment)

Test observation

Both could have problems with test observation

Sandwich Testing Strategy

Combines top-down strategy with bottom-up strategy

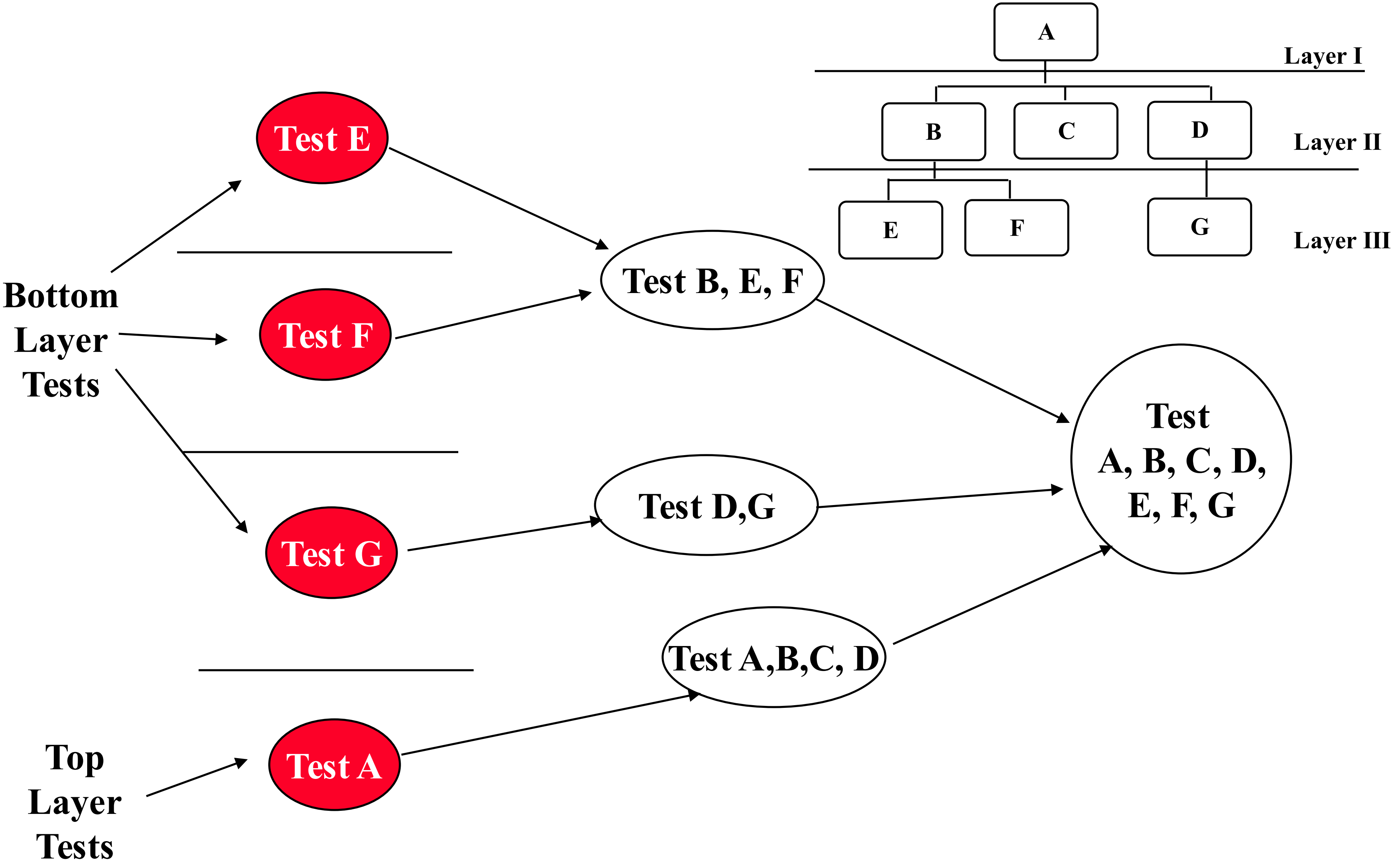
The system is viewed as having three layers

- A target layer in the middle
- A layer above the target
- A layer below the target
- Testing converges at the target layer

Selecting the target layer if there are more than three layers?

Heuristic: Try to minimize the number of stubs and drivers

Example-1


A
Layer I
B
C
D
Layer II
E
F
G
Layer III
Test E
Bottom Layer Tests
Test F
Test B, E, F
Test G
Test D,G
Test A, B, C, D, E, F, G
Test A,B,C, D
Top Layer Tests
Test A

Advantages and Disadvantages of Sandwich Testing

Top and Bottom Layer Tests can be done in parallel

Does not test the individual subsystems thoroughly before integration

Modified Sandwich Testing

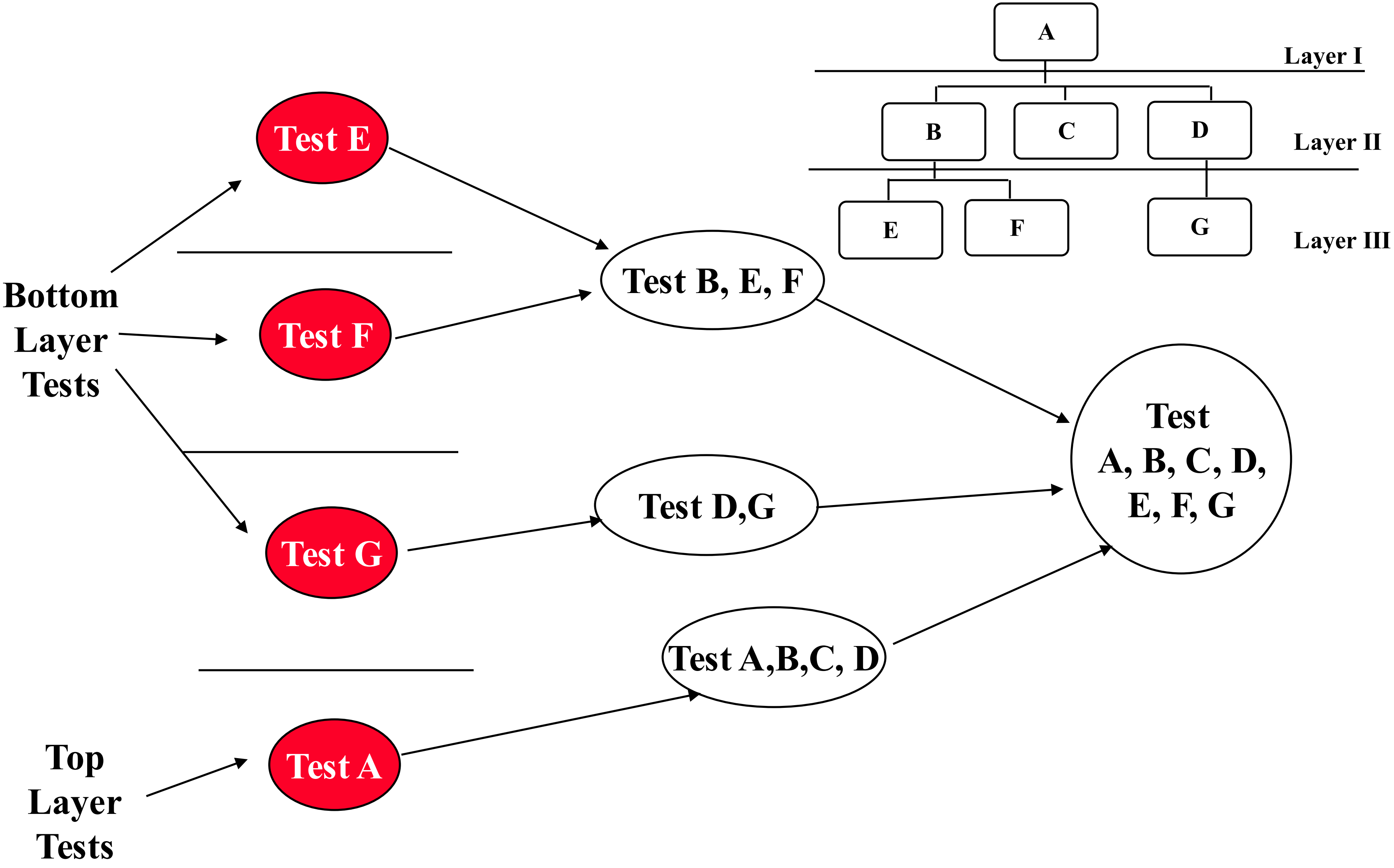
Test in parallel:

- Middle layer with drivers and stubs
- Top layer with stubs
- Bottom layer with drivers

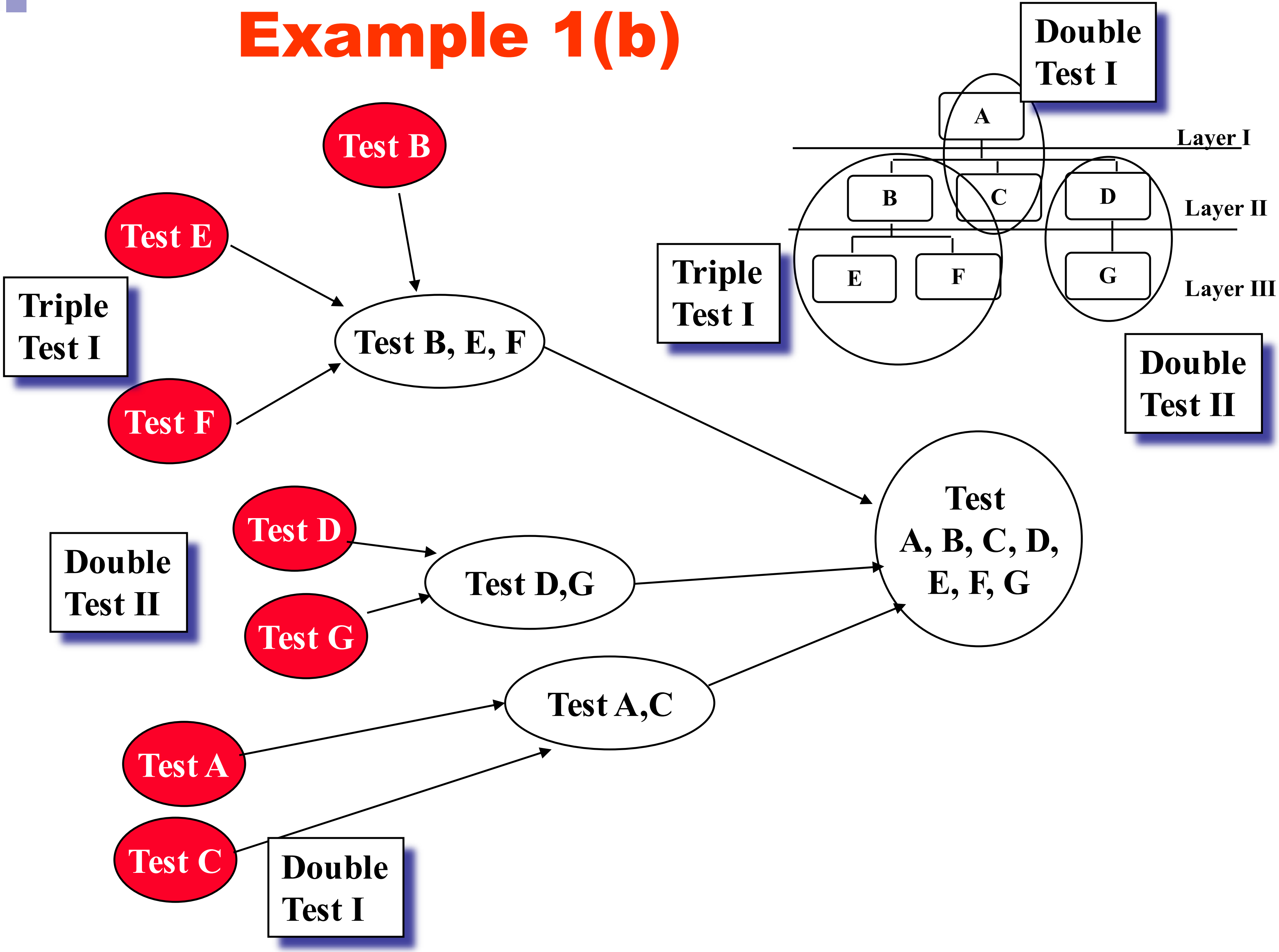
Test in parallel:

- Top layer accessing middle layer (top layer replaces drivers)
- Bottom accessed by middle layer (bottom layer replaces stubs)

Example-1 (a)


A
Layer I
B
C
D
Layer II
E
F
G
Layer III
Test E
Bottom Layer Tests
Test F
Test G
Test B, E, F
Test D,G
Test A, B, C, D, E, F, G
Test A,B,C, D
Top Layer Tests
Test A

Example 1(b)



Double Test I

A

Layer I

B

C

D

Layer II

Triple Test I

E

F

G

Layer III

Double Test II

Test B

Test E

Triple Test I

Test B, E, F

Test F

Test D

Double Test II

Test D,G

Test G

Test A, B, C, D, E, F, G

Test A,C

Test A

Test C

Double Test I

Choosing an Integration Strategy

- **Factors to consider**
  - □ Amount of test harness (stubs &drivers)
  - □ Location of critical parts in the system
  - □ Availability of hardware
  - □ Availability of components
  - □ Scheduling concerns
- **Bottom up approach**
  - □ good for object oriented design methodologies
  - □ Test driver interfaces must match component interfaces
  - □ ...
  - □ ...Top-level components are usually important and cannot be neglected up to the end of testing
  - □ Detection of design errors postponed until end of testing
- **Top down approach**
  - □ Test cases can be defined in terms of functions examined
  - □ Need to maintain correctness of test stubs
  - □ number of required stubs

Choosing an Integration Strategy

Factors to consider

- □ Number of test harness (stubs &drivers)
- □ Location of critical parts in the system
- □ Availability and stability of components
- □ Scheduling concerns

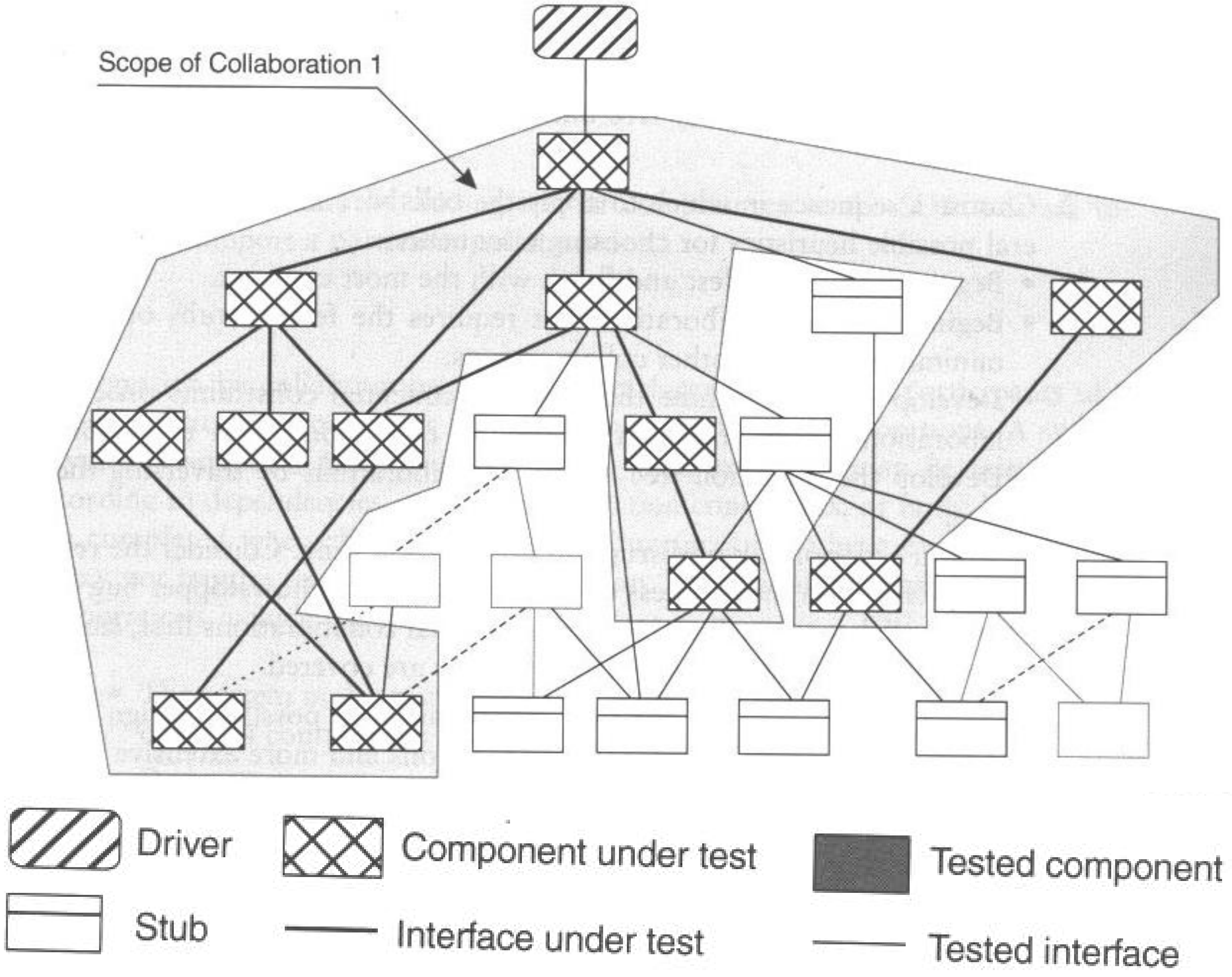
Collaboration integration

Existence of collaboration between participants

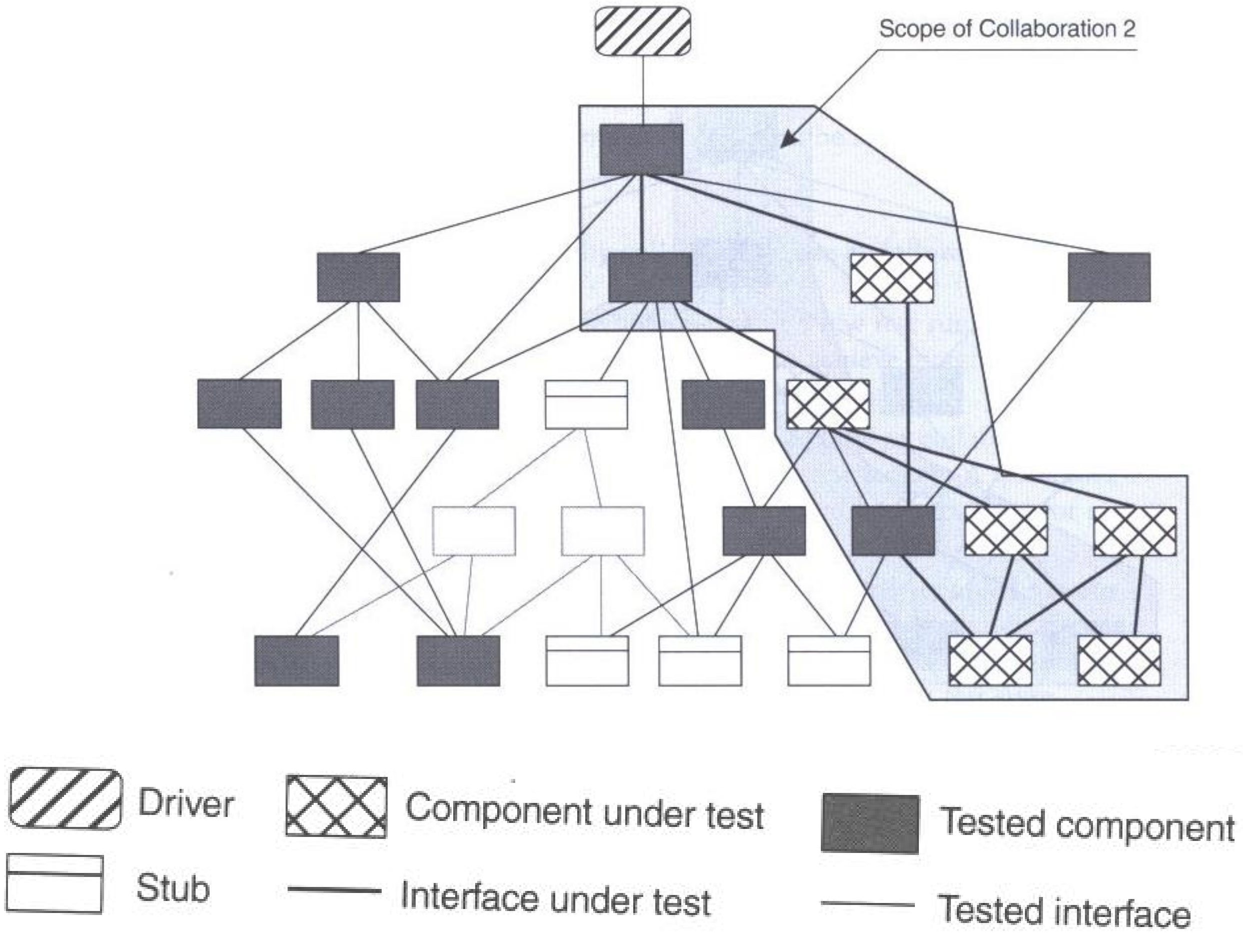
Sequence of collaborations (say, from simplest to more complex)

Participants in a collaboration are not exercised separately; a scenario-wise Big Bang integration

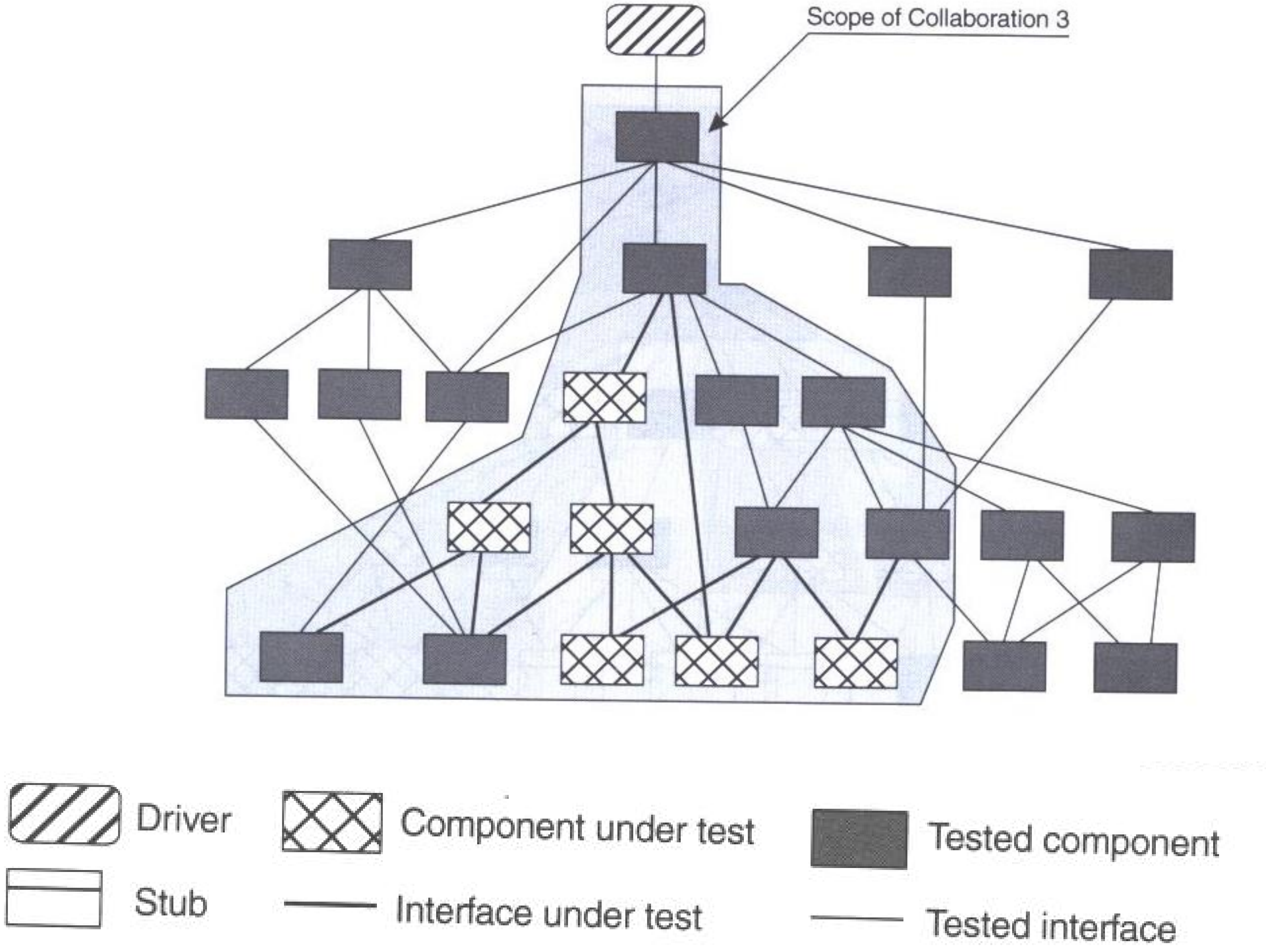
Collaboration Integration 1st configuration



Collaboration Integration 2nd configuration



Collaboration Integration 3rd configuration



Layer Integration

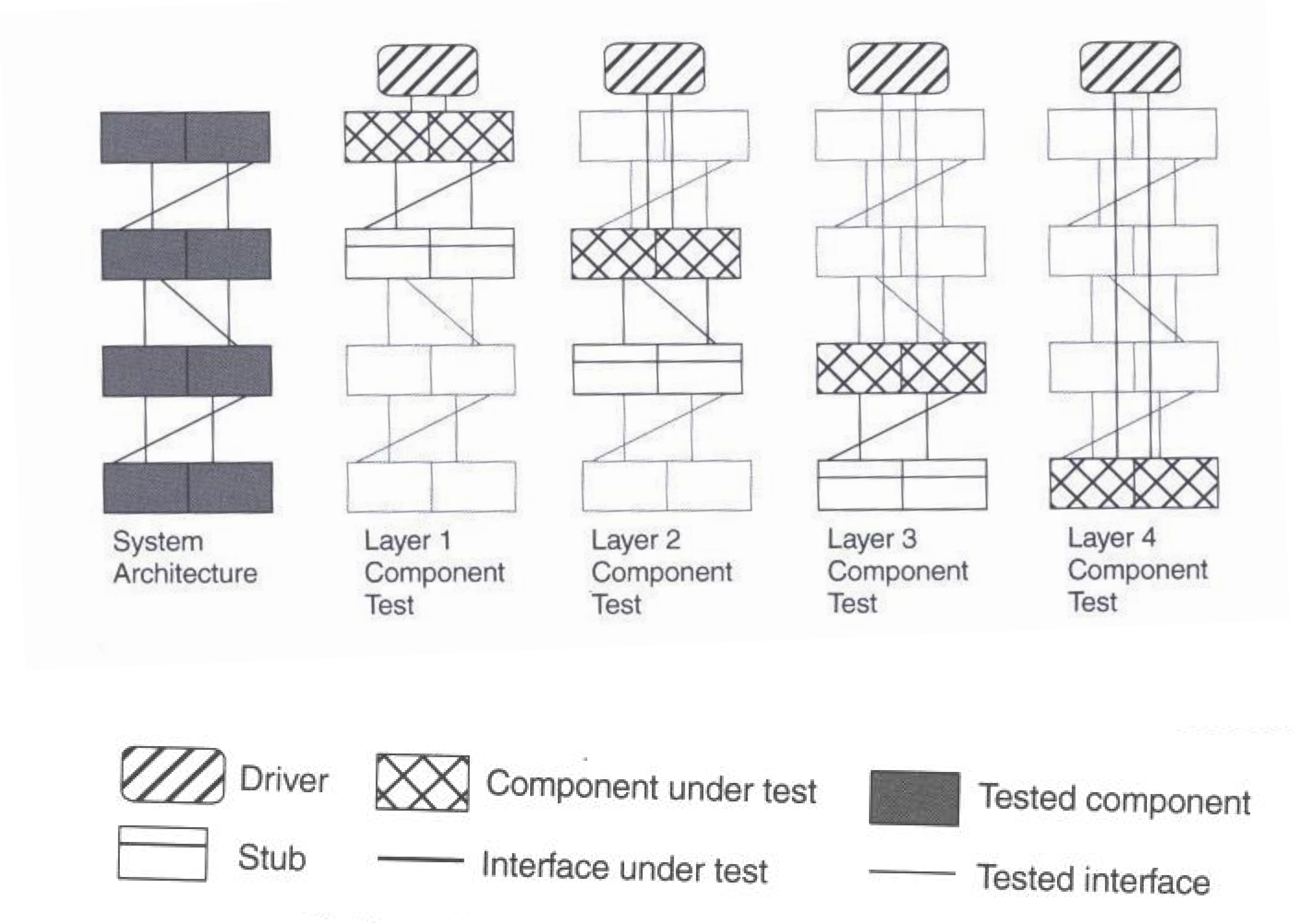
Incremental approach

layered architecture modeled as a hierarchy that allows interfaces between adjacent layers

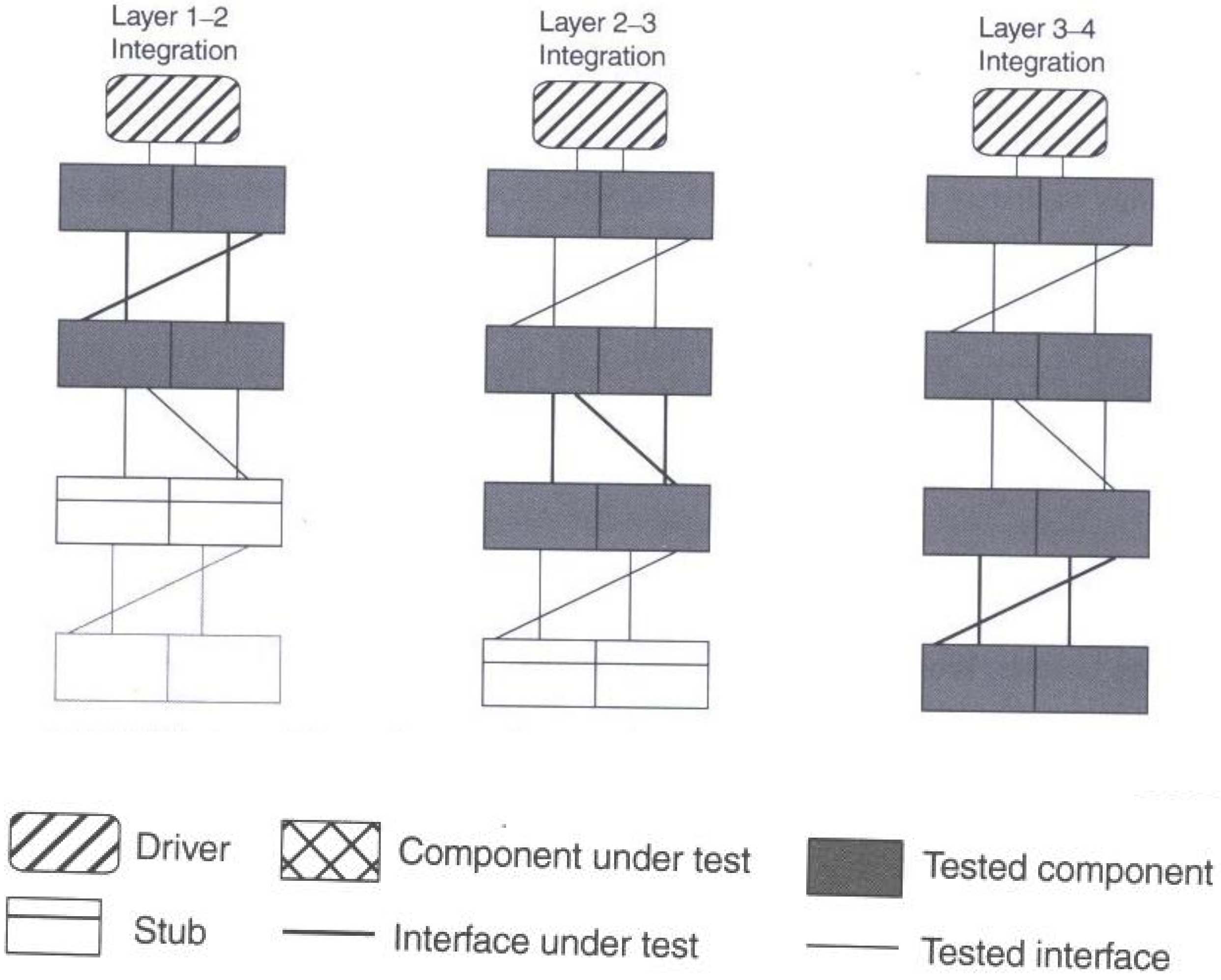
Layer integration may be top-down or bottom up

Layer integration (1)

Test each layer in isolation (consider reusable drivers)



Layer integration (2)



Module design complexity

Call graphs

Design reduction:

•start with a module’s control graph

•remove all control structures that are not involved with module calls

•use “reduced” flow graph to realize integration testing

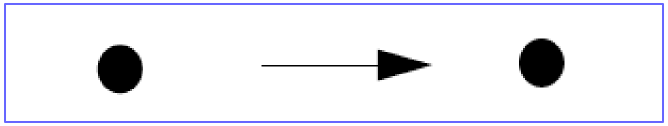
Development of a set of rules to perform design reduction

Design reduction – rules (1)

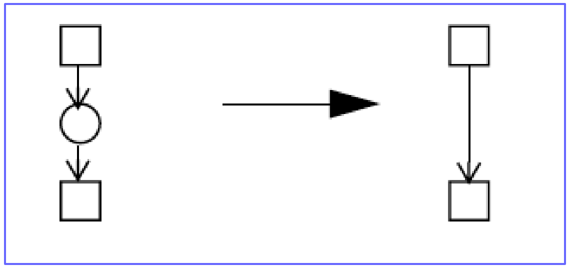
Eliminate parts of flow graph not involved with module calls

● = Call node ○ = Non-call node
⊕ = Path of zero or more non-call nodes
□ = Any node, call or non-call

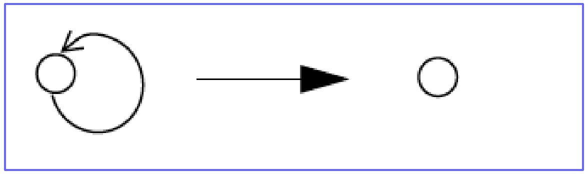
Call rule



1. Sequential rule
eliminates sequences of non-call nodes



2. Repetitive rule
eliminates test loops not involved with module calls

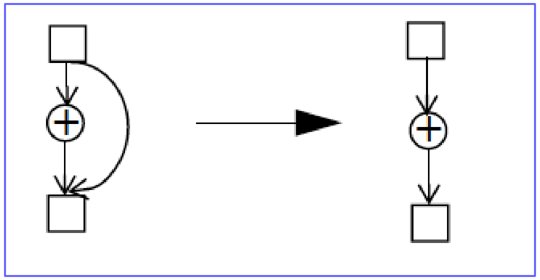


Design reduction – rules (2)

● = Call node ○ = Non-call node
⊕ = Path of zero or more non-call nodes
□ = Any node, call or non-call

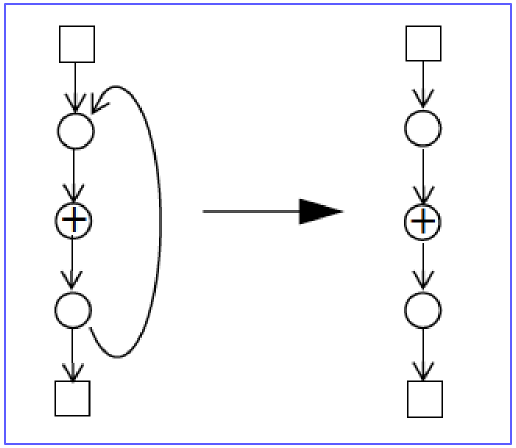
3. Conditional rule

eliminates conditional statements that do not contain calls in their bodies



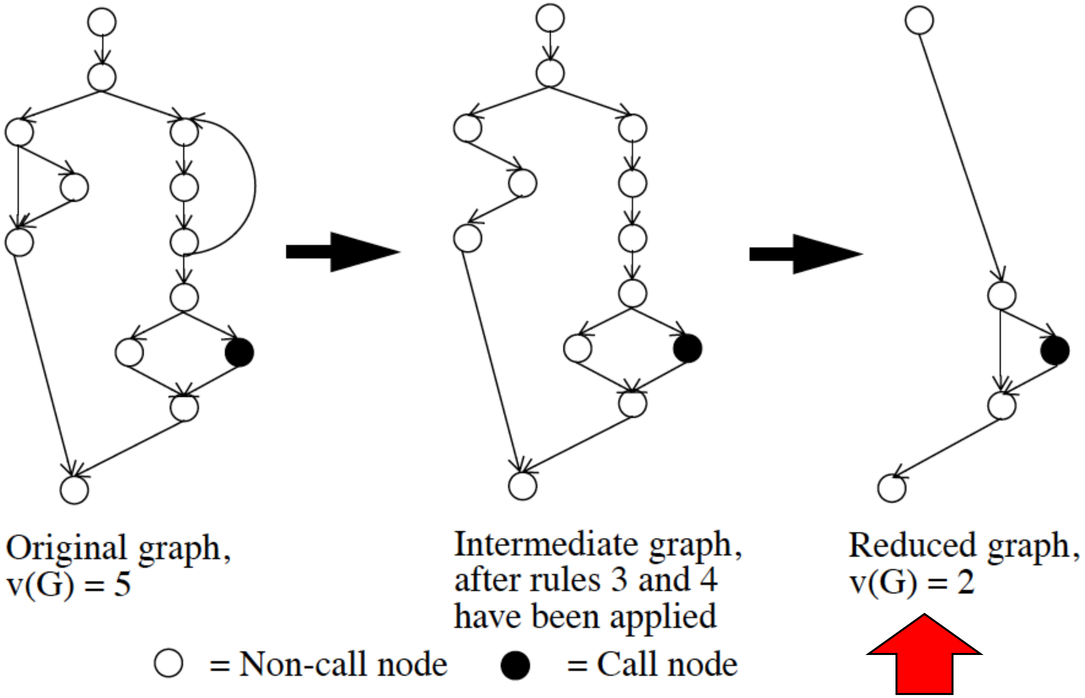
4. Looping rule

eliminates bottom test loops not involved with module calls



Design reduction – example

Apply the rules iteratively until none of them can be applied – design reduction has been completed



Original graph,
v(G) = 5

Intermediate graph,
after rules 3 and 4
have been applied

Reduced graph,
v(G) = 2

○ = Non-call node ● = Call node

module design complexity iv(G) – cyclomatic complexity of the reduced graph

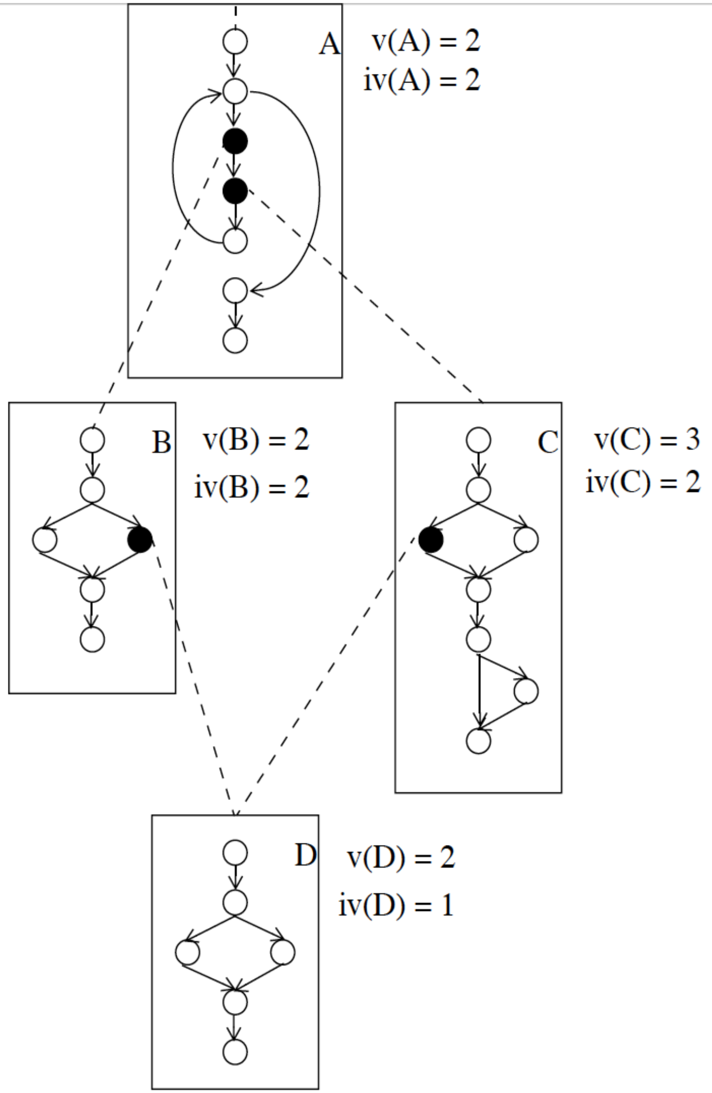
Integration complexity

n modules (G_1, G_2, …, G_n) with module design complexities iv(G_1), iv(G_2),…, iv(G_n)

Integration complexity S: number of independent integration tests through a program design

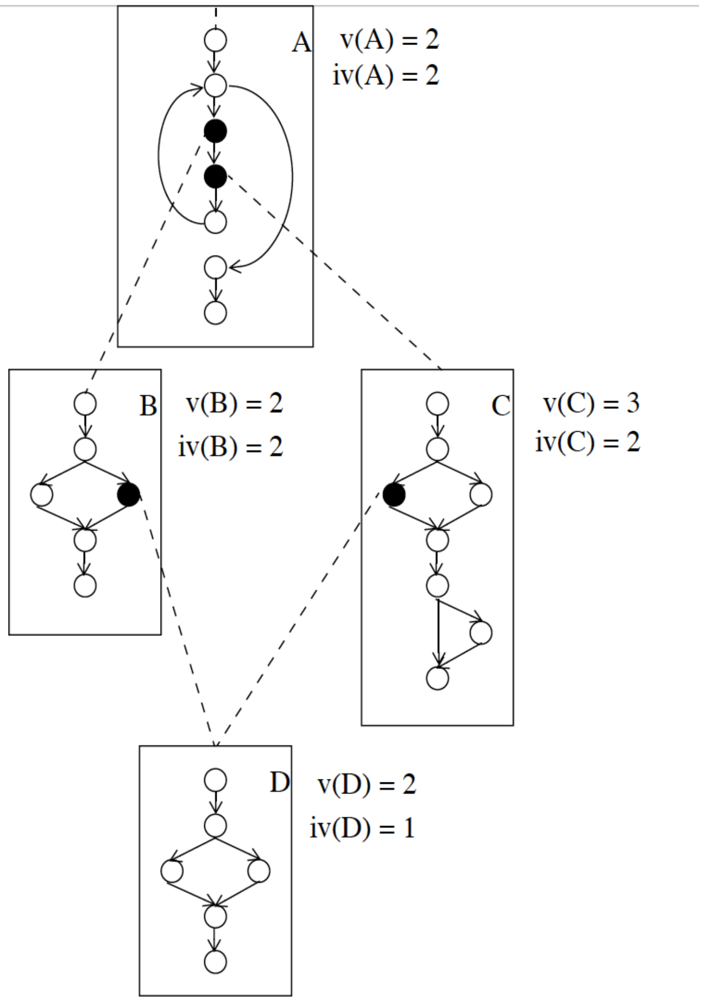
$$S = \mathop{\square}_{i=1}^{n} iv(G_i) - n + 1$$

Integration complexity-example



S=(2+2+2+1)-4+1=4

Integration complexity-example



Independent integration tests

X→Y←X: X calls Y which returns to X

1. A
2. A→B←A→C←A
3. A→B→D←B←A→C←A
4. A→B←A→C→D←C←A