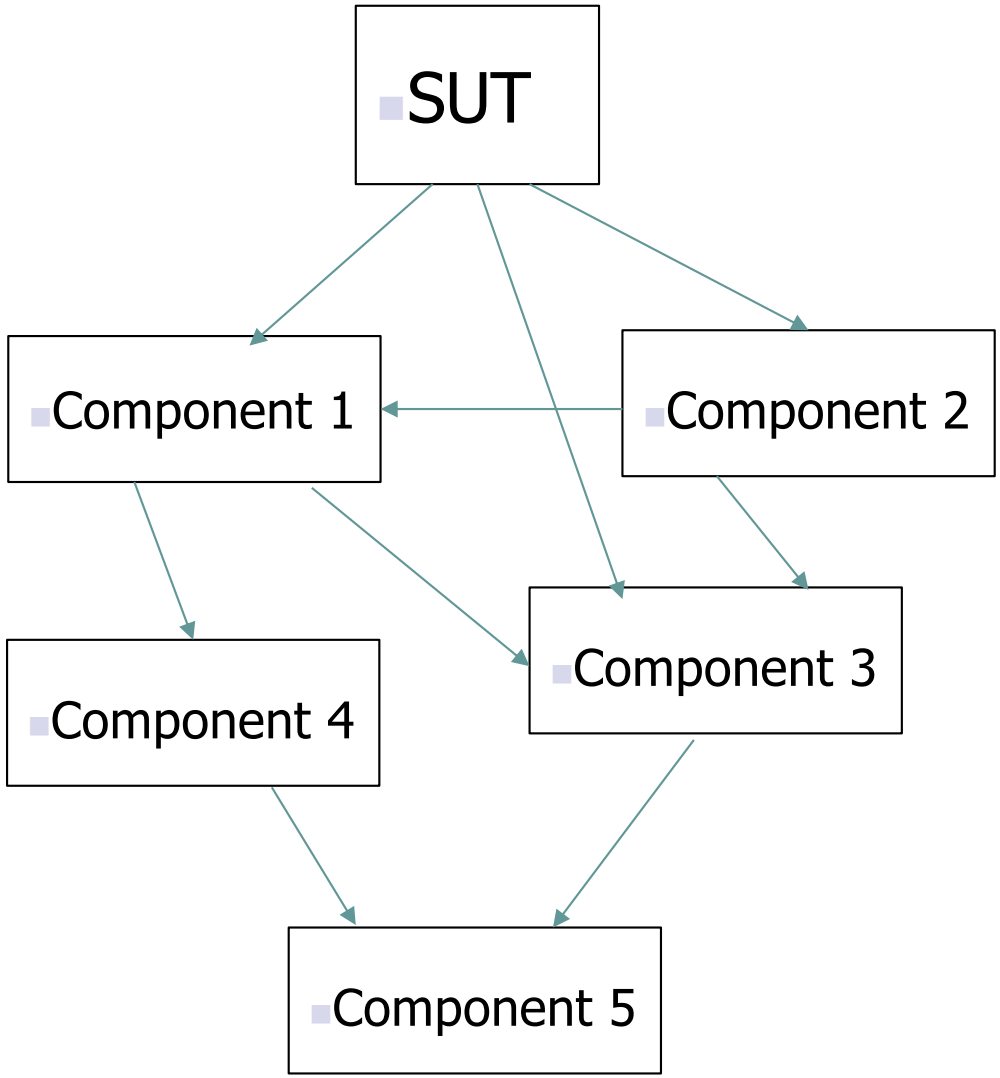
Integration Testing

Integration testing



- Requirements for testing at system scope
  1. Each component was tested isolated
  2. The communication among components and with the SUT was also tested
     - Interface misusage
     - Interface misunderstanding

Integration testing

- Software systems are built with components that must interoperate

- Primary purpose:
  - To reveal component interoperability faults so that testing at system scope may proceed with the fewest possible interruptions

- An integration strategy must answer three questions
  - Which components are the focus of integration test?
  - In what sequence will component interfaces be exercised?
  - Which test design technique should be used to exercise each interface?

Integration testing = a search for component faults that cause intercomponent failures

When?

- Integration testing execution cannot be started until the modules to integrate are completely created and tested in **isolation**
- Should not happen at the end of the development cycle
- Should be done in parallel with the development.

Integration in OO Systems

- Integration in object-oriented development takes place at all scopes
  - Within a class
  - Within a class hierarchy
  - Between a client and its servers
  - Within a cluster of related classes
  - Within a subsystem
  - Within an application system

Integration in OO Systems - 2

System (Scope of integration)	Component (Focus of integration)	Typical intercomponent interface (Location of integration faults)
Class	Method	Instance variables Intraclass messages
Cluster	Class	Interclass messages
Subsystem	Cluster	Interclass messages Interpackage messages
System	SubSystem	Interprocces communication Remote procedure call

Integration faults

- Integration testing can reveal component faults that cause failures when components interact
- Typical interface bugs include the following
  - Configuration/version control problems
  - Missing, overlapping, or conflicting functions
  - The client send a message that violates the server's precondition or sequential constraints
  - Wrong object bound to message (polymorphic target)
  - Wrong parameters, or incorrect parameter values
  - Incorrect usage of virtual machine, ORB, or OS services
  - …
- Can be classified as:
  - Interface misuse
  - Interface misunderstanding

Integration testing strategies - 1

- Big Bang Integration: Attempt to demonstrate system stability by testing all components at the same time
- Bottom-up Integration: Interleave component and integration testing by following usage dependencies
- Top-down Integration: Interleave component and integration testing by following application control hierarchy
- Sandwich Integration: Use Top-down and Bottom-up
- Collaboration Integration: Choose an order of integration according to collaborations and their dependencies
- Layer Integration: Incrementally exercises interfaces and components in a layered architecture

Integration testing strategies - 2

- Backbone Integration: Combine Top-down Integration, Bottom-up Integration and Big Bang Integration to reach a stable system that will support iterative development
- Client/Server Integration: Exercise a loosely coupled network of components, all of which use a common server component
- Distributed Services Integration: Exercise a loosely coupled network of peer-to-peer components
- High-frequency Integration: Develop and rerun an integration test suite test hourly, daily or weekly

Scope-specific considerations

- Classes
  - The system is the class under test
  - The components to be integrated are CUT methods, superclass methods, instance variables, and parameters in messages received and sent by the CUT
  - Testing of class responsibilities is so closely coupled to class integration that it does not make sense to treat the two as separate test design patterns

Cluster-specific Considerations

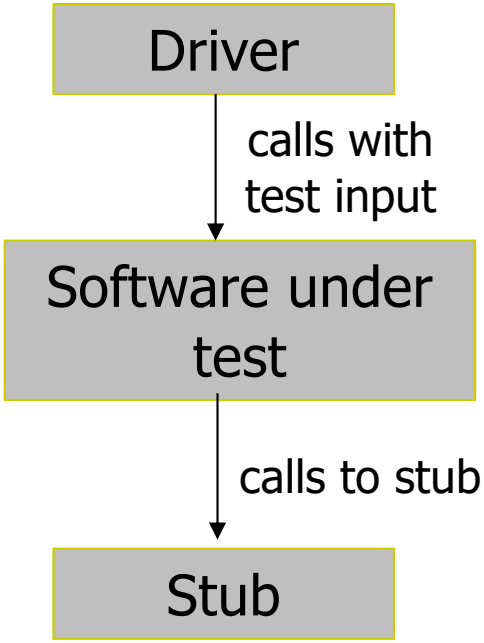
- The components to be integrated are server objects used by the CUT
  - Bottom-up Integration is the most widely used technique
  - Top-down Integration is possible if one class is the head of the cluster
  - Collaboration Integration of a modal class may be accomplished with ***Class* Association Test**, **Round-trip Scenario Test**, or **Mode Machine Test**
  - If the class under test catches exceptions, **Controlled Exception Test** may be useful
  - *Big Bang* integration is not recommended
  - There is no direct analogue to the other patterns

System-specific considerations - 2

- Subsystem/system
  - The scope of an implementation under test is often larger than of a single class or small cluster
- Bottom-up Integration works well for small to medium systems
- Top-down, Collaboration, and High-frequency Integration work for almost any scope or architecture
- Layer Integration applies to layered architectures
- Client/Server Integration is appropriate for client/server architectures
- Distributed Services Integration is appropriate for decentralized networks containing peer-to-peer nodes
- Backbone Integration is well suited to small to medium systems and especially useful for embedded systems
- Big Bang Integration is useful in a few limited circumstances

Drivers and stubs

- **Driver:** A program that calls the interface procedures of the module being tested and verifies the results
  - A driver simulates a *client* that calls the SUT
  - Corresponds to the execution of a test suite
- **Stub:** A program that has the same interface procedures as a module that is being called by the module being tested but is simpler.
  - It simulates a module called by the SUT
  - It is temporary or dummy software that is required by the SUT for it to operate properly
    - In the context of the driver



Stub

- Usually, it is very simple
- It must have the **exactly** the same interface as the module being stubbed
- And **behave** as the module being stubbed for the inputs used
- Very brittle
- Or can use a mock framework

```
int getPIN(int cardNumber) {
  if (cardNumber == 123456789)
     return 1234;
  else if (cardNumber == 123456788)
     return 1238;
  else if (cardNumber == 123456789)
     return 4321;

    throw new InvalidCardException();
}
```

Finding a bug and locate the cause of bug

- Unit Testing
  - *Easy* to locate cause since scope is usually narrow
- Integration Testing
  - Harder to locate cause
    - Each test case may involve several components
    - It is hard to locate the faulty component
      - Can apply incremental integration
    - And understand the cause of failure
      - Can use logging to have more information

Examples of integration faults - 1

Ex2mples of integration faults - 2

Big Bang Integration

- Intent
  - Demonstrate stability by attempting to exercise an entire system with a few test runs

- Context
  - The SUT is stabilized and only a few components have been added or changed since the last time it passed a system scope test suite
  - The SUT is small and testable, and each of its components has passed adequate component scope tests
  - When components are so tightly coupled that they can not be exercised separately

Fault model and Strategy

- Fault Model
  - The fault model is ambiguous and opportunistic
  - The hope is that the system will “run” and thereby demonstrate that system testing can begin

- Strategy
  - Dispenses incremental integration testing
  - The entire system is built, and a test suite is applied to demonstrate minimal operability at system scope
  - Test suit may be developed at system scope by using an appropriate responsibility-based test design pattern

Big Bang integration

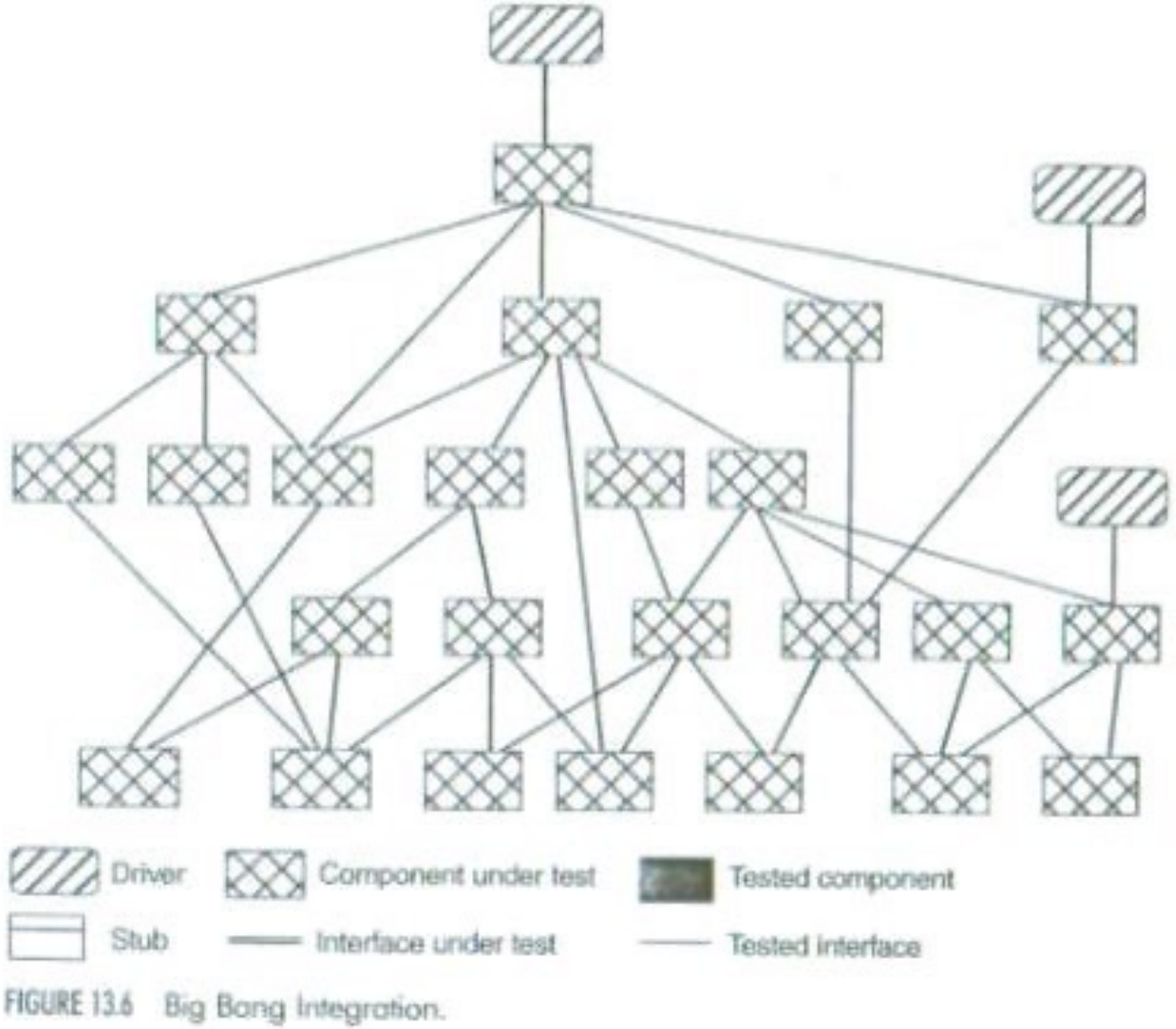


FIGURE 13.6 Big Bang Integration.

Entry and exit criteria

- Entry criteria
  - All components have passed component scope testing
  - A physical, functional, and environmental audit has been conducted and has not found any anomalies that would interfere with integration testing
- Exit criteria
  - The test suite passes

Consequences

- Consequences
  - Debugging can be difficult because you receive fewer clues about fault locations
  - Even if SUT passes, many interface faults can hide and waylay subsequent system scope testing
  - Integration testing cannot be started until all modules are completely created and tested
  - Under favorable circumstances it can result in quick completion of integration testing
  - Few (if any) integration drivers or stubs are developed
- Known Uses
  - It is used in Backbone Integration

Incremental integration testing

- Incremental integration is the most effective technique
  - Add components a few at a time and then test their interoperability
  - Advantages:
    - Interfaces are systematically exercised
    - Observed failures are most likely to come from the most recently added component

- Strategy
  - **How to define the integration sequence?**
  - A sequence of components should be identified using careful analysis of component dependencies
  - Integration testing must be planned and managed to follow these dependencies

Dependency analysis

- Components typically depend on each other in many ways
- Class and cluster scope dependencies result from explicit binding mechanisms
  - Composition and Inheritance
  - Global variables; Server objects
  - Objects used as message parameters
  - ...
- Similar intercomponent dependencies occur at subsystem and system scopes
  - Components can communicate
- Dependencies often dictate the sequence of testing

Integration Patterns

- Dependency tree that will be used to show successive configurations

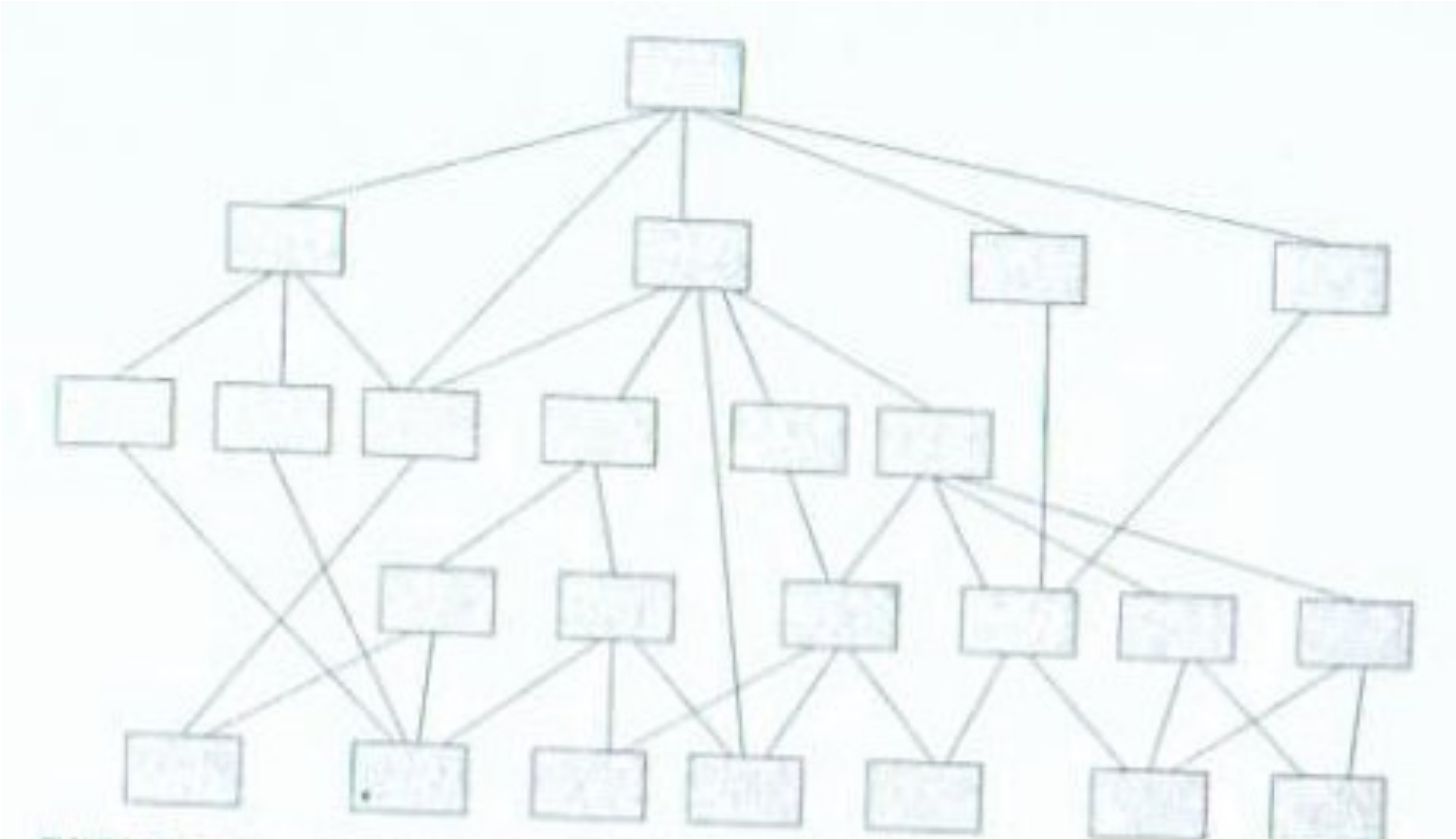


FIGURE 13.5 Generic dependency tree.

Bottom-up Integration

- Intent
  - Demonstrates stability by adding components to the SUT in uses-dependency order, beginning with components having the fewest dependencies
- Context
  - Components with the least number of dependencies are tested first
  - When these components pass, their drivers are replaced with their clients; another round of testing then begins
  - It is often used to support unit scope testing in the iterative and incremental development of a subsystem's components

Strategy

- Test Model
  - Dependency tree
  - Responsibility for each component
- Test Procedure
  - For (n = leaf level; n <= root level; n++)
    - Code all components of level n
    - Develop responsibility test suite for each component in level n
      - may need to add extra test cases. Why?
    - Exercise test suites of level n

Bottom-up Integration

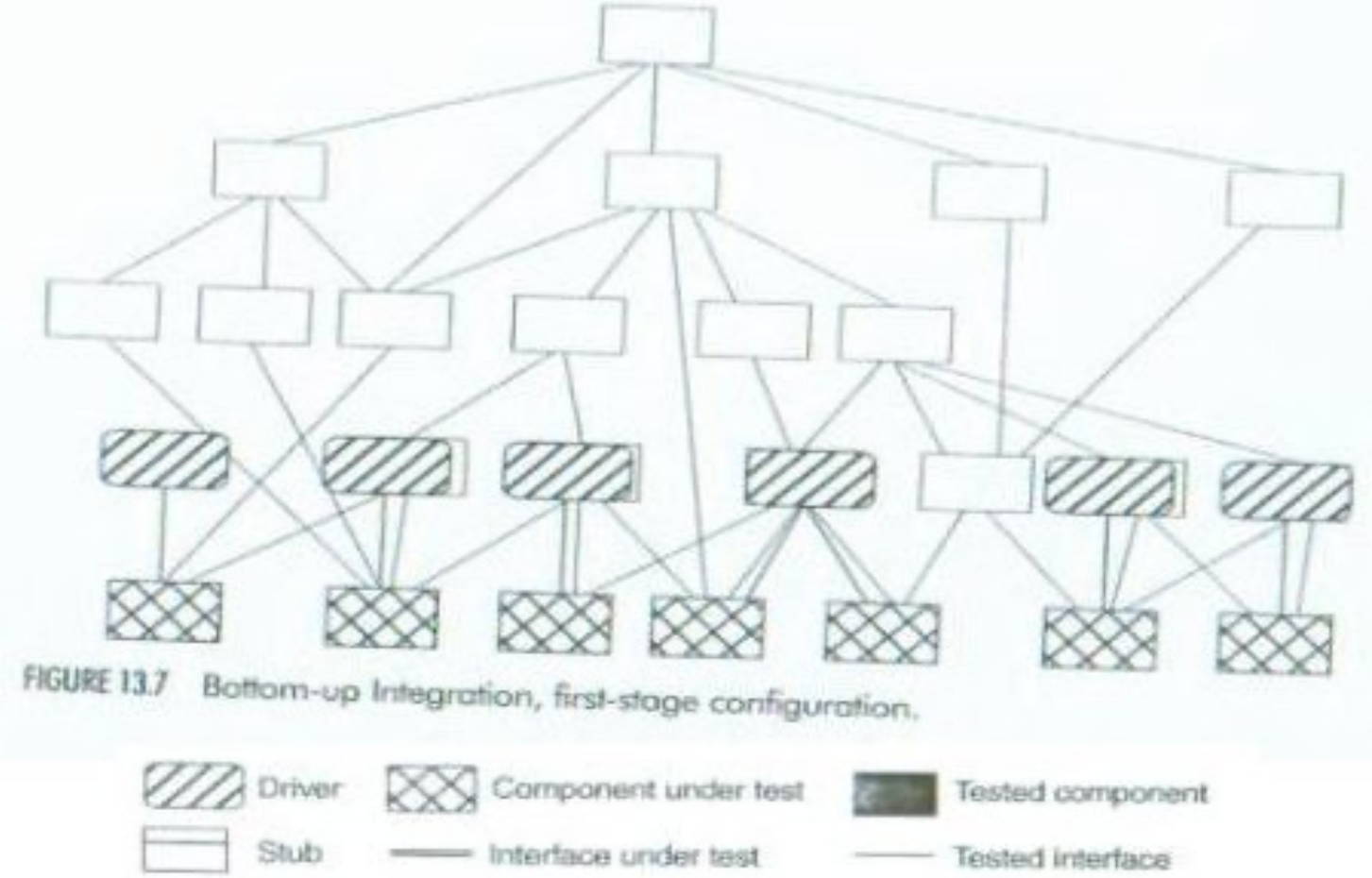


FIGURE 13.7 Bottom-up integration, first-stage configuration.

Bottom-up Integration

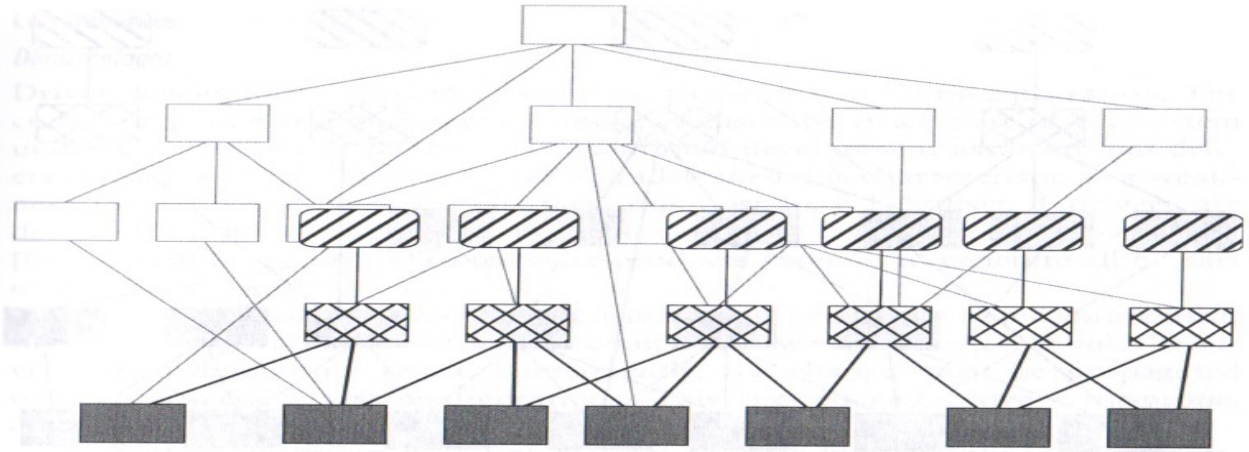


FIGURE 13.8 Bottom-up Integration, second-stage configuration.

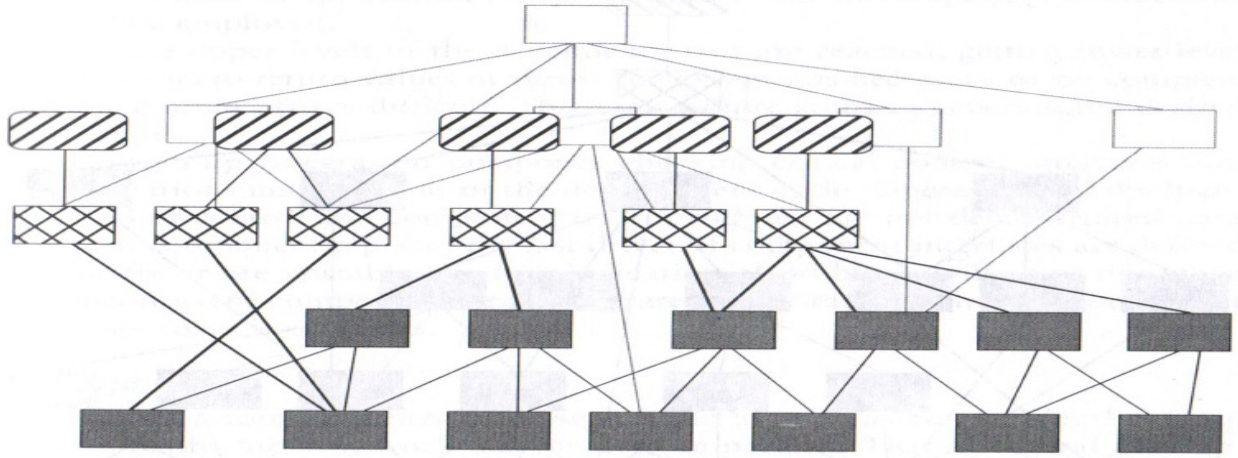


FIGURE 13.9 Bottom-up Integration, third-stage configuration.

Bottom-up Integration

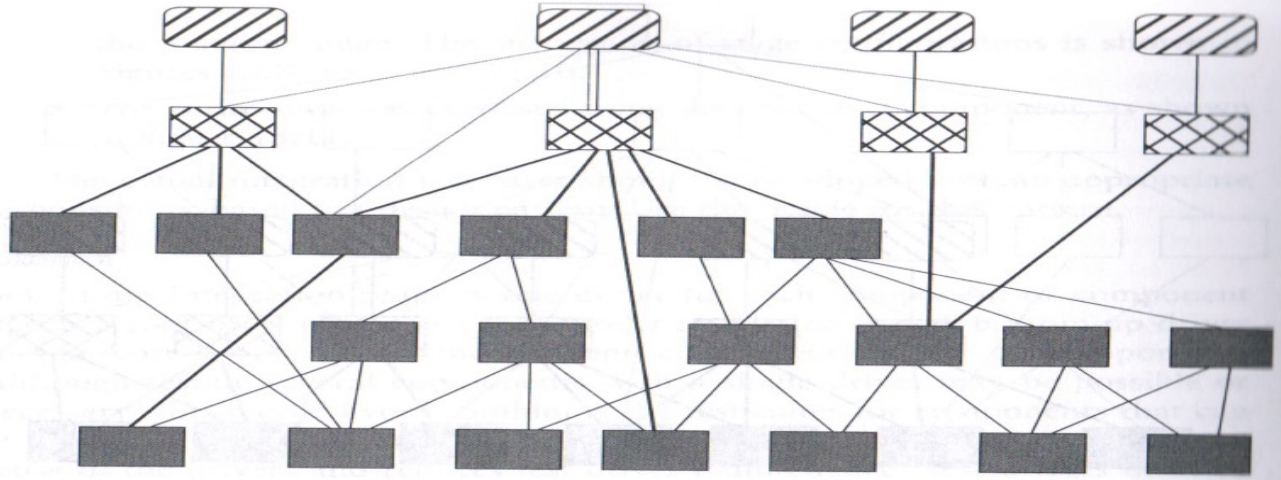


FIGURE 13.10 Bottom-up Integration, fourth-stage configuration.

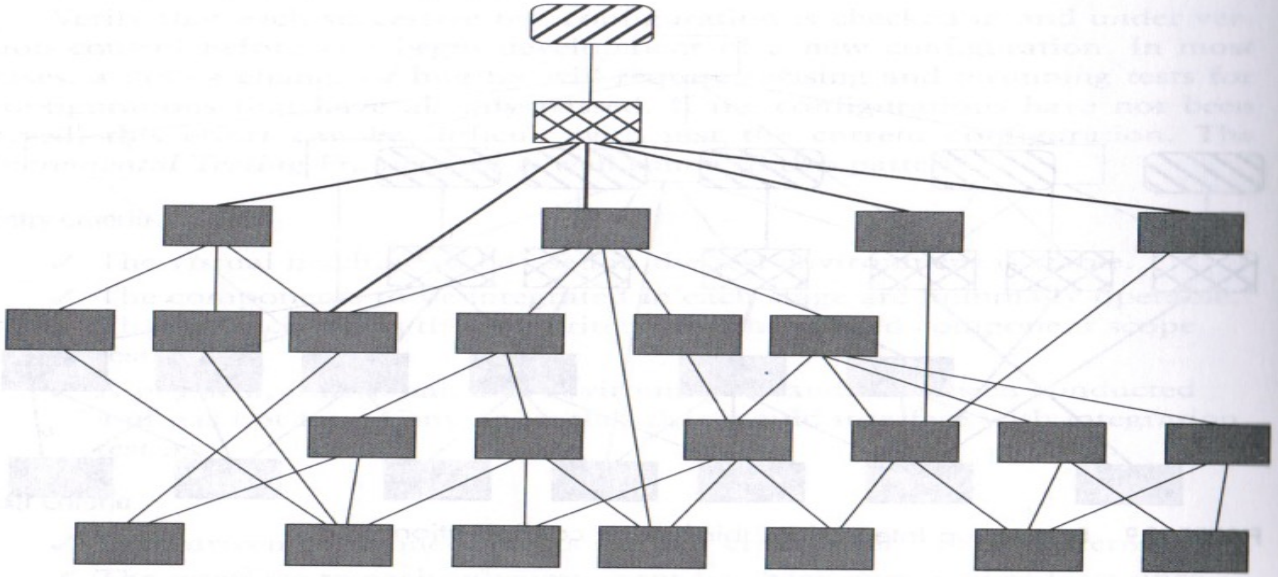


FIGURE 13.11 Bottom-up Integration, final configuration.

Bottom-up Integration

- Automation
  - It is necessary one driver for each component or component that is the root of a subtree in the dependency relationship
  - The drivers may be revised as the class under test is revised

Entry and exit criteria

- Entry Criteria
  - The virtual machine to be used in the test environment is stable
  - The components to be integrated in each stage are minimally operable
- Exit Criteria
  - Each driver component meets the exit criteria for its test pattern
  - The interface to each component has been exercised at least once
  - Integration testing is complete when all root-level components pass their test suites

Consequences

- Disadvantages
  - Driver development is the most significant cost
  - The driver does not directly exercise intercomponent interfaces
  - Postpones checking critical control interfaces and collaborations until the end of the development cycle
- Advantages
  - May begin as soon as any leaf-level component is ready
  - Work may proceed in parallel
  - Although this pattern reduces stubbing, stubs may still be needed to break a cycle or simulate exceptions
  - It is suitable to responsibility-based design

Top-down integration

- Intent
  - Demonstrate stability by adding components to the SUT in control hierarchy order, beginning with the top-level control objects
- Context
  - Control objects typically implement essential and nontrivial control strategies and therefore present relatively high risk
  - Top-down integration focuses on control components first, making the demonstration of system scope end-to-end operability a high priority

Strategy: Test Model

- The apex of control may be represented in a Collaboration Diagram, Sequence Diagram or Statechart of the SUT
- A responsibility test suite may be developed with any appropriate test design pattern
  - *Modal Class Test* or *Mode Machine Test*
  - *Collaboration Integration*, *Round-trip Scenario Test* or *Covered in CRUD*

Strategy: Test Procedure

- Model the control hierarchy as a dependency tree
- Develop a staged plan for implementation and testing
- Design a responsibility-based test suite at system scope
  1. Develop and test the component(s) at the highest level of control first
  2. Continue in breadth-first swath at each level, replacing the server stubs with a full implementation
     1. One step or several?
        1. Risk of a small bang of many components in one level
        2. Add a few each time instead
  3. Continue in this manner until all servers in the SUT have been implemented and exercised

Top-down Integration – Step 1 and 2

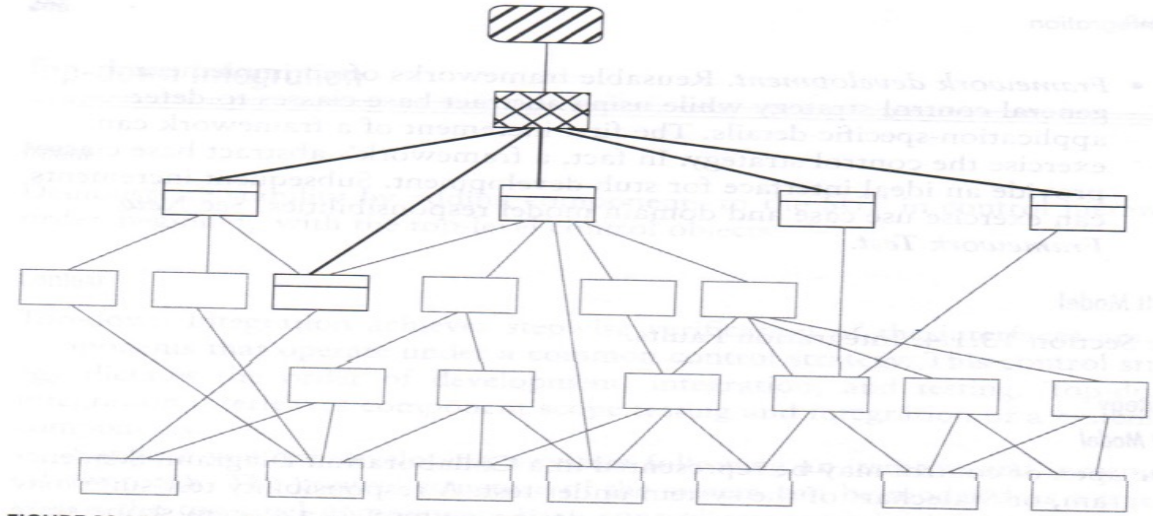


FIGURE 13.12 Top-down Integration, first-stage configuration.

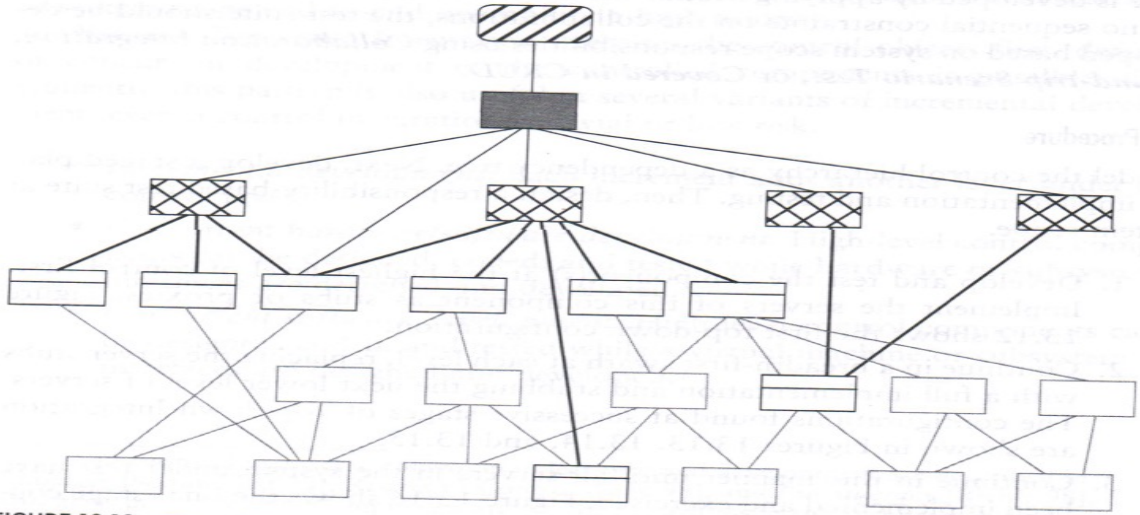


FIGURE 13.13 Top-down Integration, second-stage configuration.

Top-down Integration – Step 3 & 4

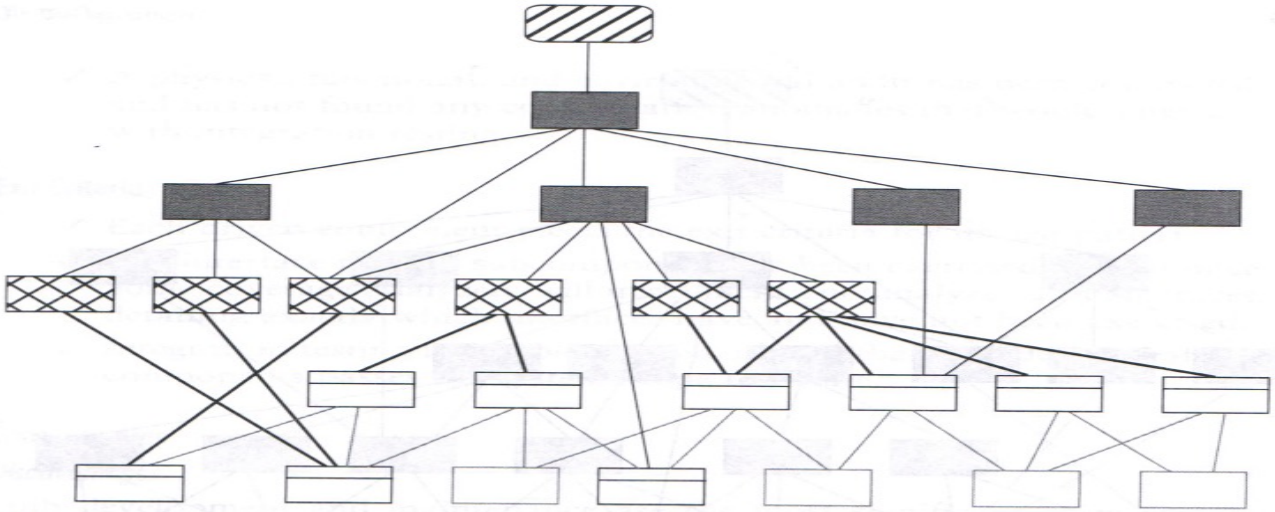


FIGURE 13.14 Top-down Integration, third-stage configuration.

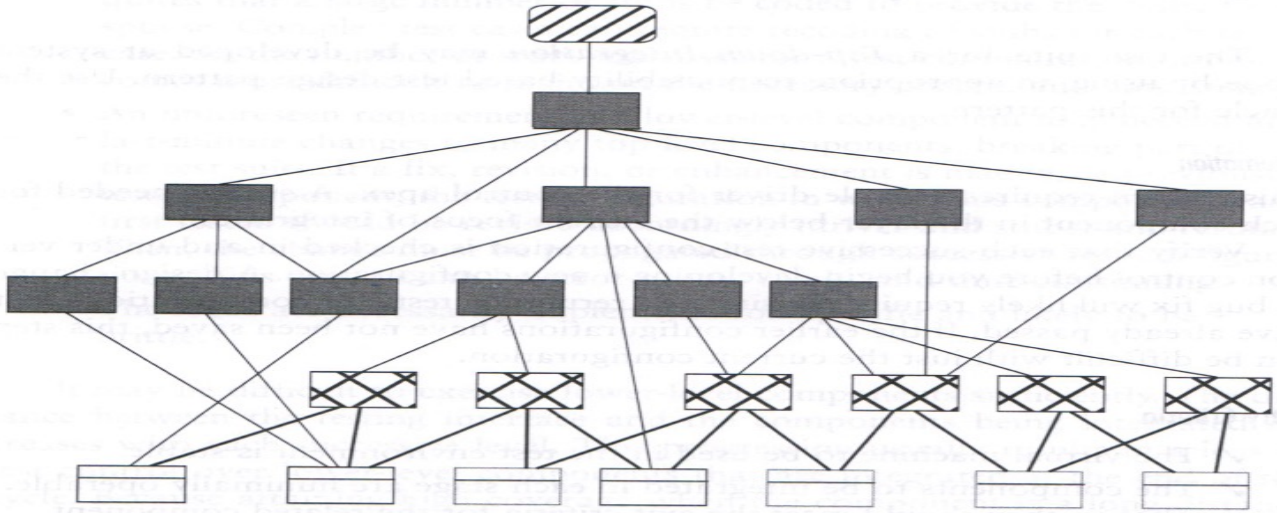


FIGURE 13.15 Top-down Integration, fourth-stage configuration.

Top-down Integration – Step 5

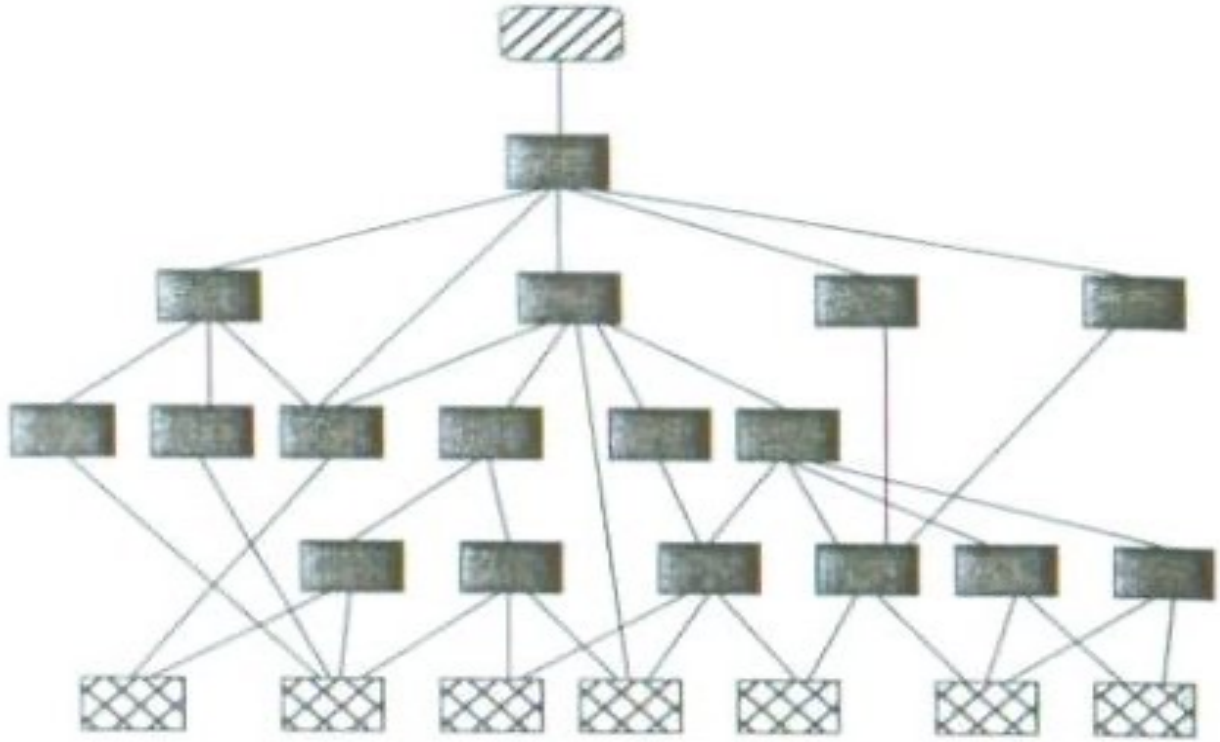
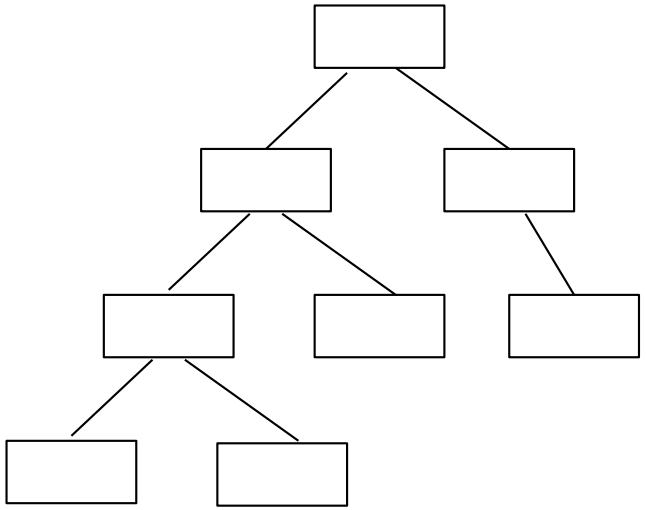


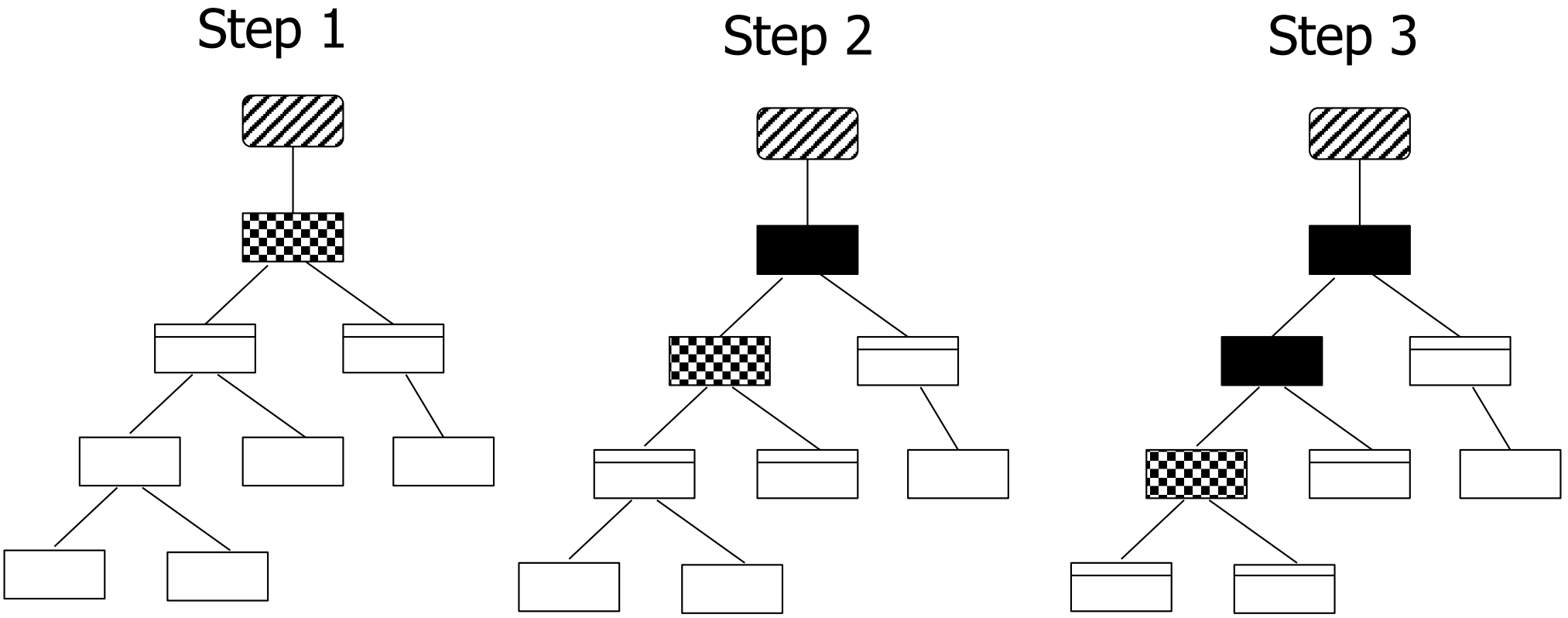
FIGURE 13.16 Top-down integration, final-stage configuration.

Top-down Integration: Depth-first Approach

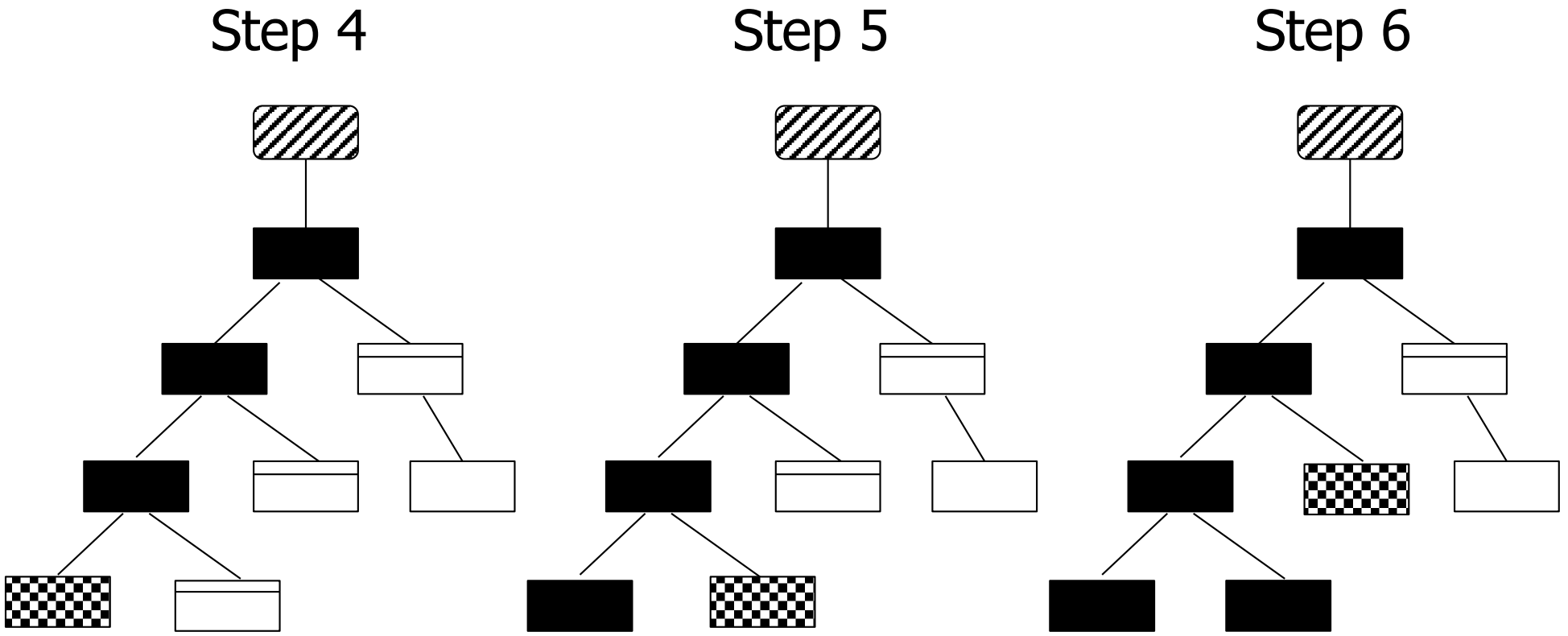
- It is also possible to apply a depth-first approach instead of breadth-first
- In depth first approach all modules on a control path are integrated first
- In this case, we just replace one stub in each iteration
- Example:



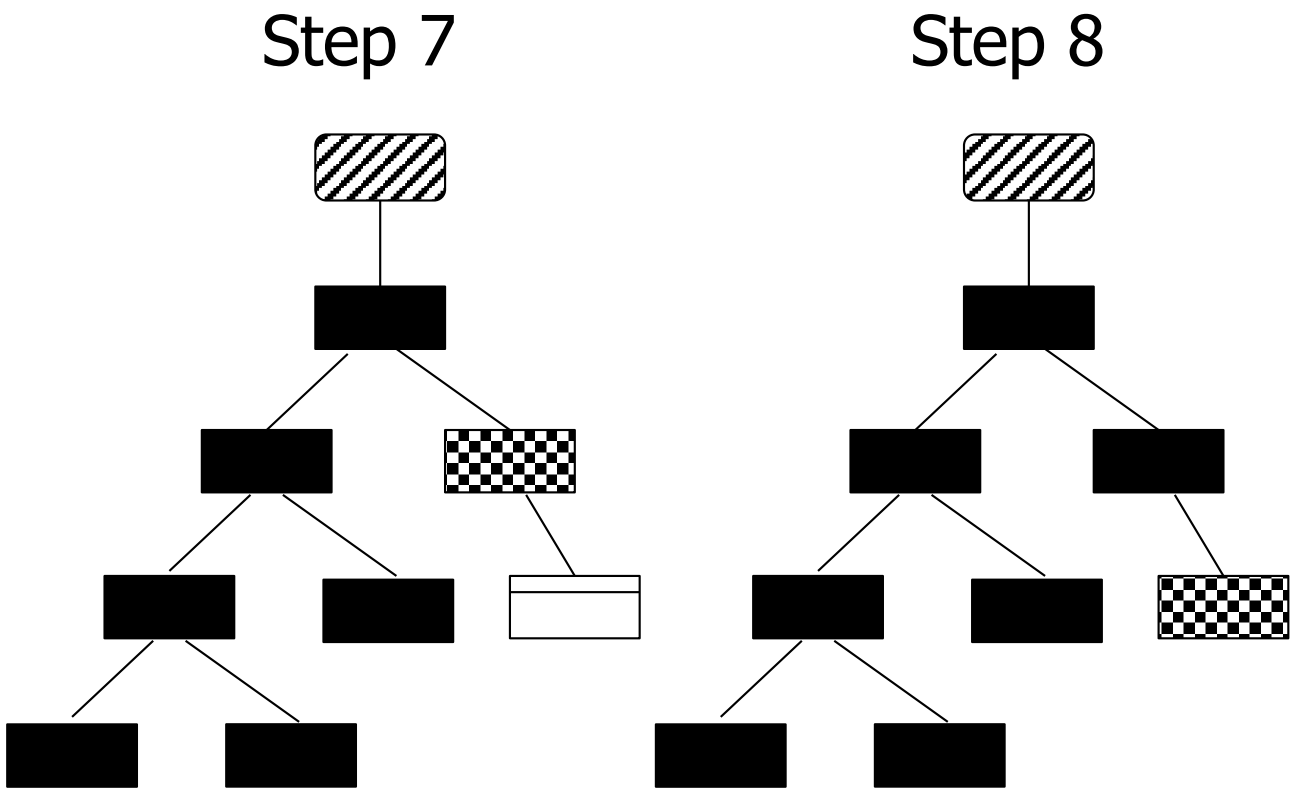
Top-down Integration: Example 1



Top-down Integration: Example 2



Top-down Integration: Example 3



Automation

- Requires a single driver for the control apex
- A stub is needed for each component in the layer below the current focus of integration

Entry and Exit Criteria

- Entry Criteria
  - The virtual machine to be used in the test environment is stable
  - The components to be integrated in each stage are minimally operable
- Exit Criteria
  - Each driven component meets the exit criteria of its test pattern
  - The interface to each subcomponent has been exercised at least one
  - Integration testing is complete when a build that includes all leaf-level components passes the system scope test suite

Consequences

- Disadvantages
  - Setting up a test requires that a large number of stubs be coded to provide the desired response
  - An unforeseen requirement in a lower-level component may necessitate last-minute changes to many top-components, breaking part of the test suite (or the design is not optimal)
  - The stubs are necessarily implementation-specific and likely to be brittle
  - It may be difficult to exercise lower-level components sufficiently
  - Can have a small bang when integrating the components of a layer
- Advantages
  - Control components are developed and integrated early
  - Testing and integration may begin early
  - The cost of driver development is reduced
  - Components may be developed in parallel with testing

Sandwich Integration Testing

- Intent
  - Demonstrate stability by adding components incrementally using both Top-down and Bottom-up approaches
- Context
  - It is a combination of both Top-down and Bottom-up integration testing

Test Procedure

- Select a layer in the dependency tree
  - Designated as the **target** layer
  - Usually near the *middle*
- Integrate the components incrementally
  - The layers above the target are integrated using the top-down approach
  - The layers bellow the target are integrated using the bottom-up approach
  - Testing converges at the target layer

Consequences

- Integration can be faster
  - Top and bottom layers can be tested in parallel
- Can reduce the need for stubs and drivers
- However
  - It can be more complex to plan
  - Selecting the ‘best’ target layer can be difficult if there are more than 3 layers
    - Heuristic: Try to minimize the number of stubs and drivers

Layer Integration

- Intent
  - Uses an incremental approach to verify stability in a layered architecture
- Context
  - For use with a pure layer architecture system
- Fault Model
  - Same as with general integration faults
  - Inadequate performance may present a critical failure in systems such as device drivers

Strategy: Test model

1st: Develop each layer in isolation

- Identify layers and suitable integration test patterns for each
- Combines top-down and bottom-up integration
- Usually
  - Top layers are tested top-down
    - Use Mode Machine, Round-trip Scenario or Collaboration Integration
    - Use stubs for layer below
  - Middle and bottom layers developed bottom-up

2nd: Do Layer integration

- Layer integration may be top-down or bottom-up
- Top-down:
  - Perform top-down integration of layers
- Bottom-up:
  - Perform bottom-up integration of layers

Strategy: Test Procedure – 1st part

- First: Test each layer in isolation

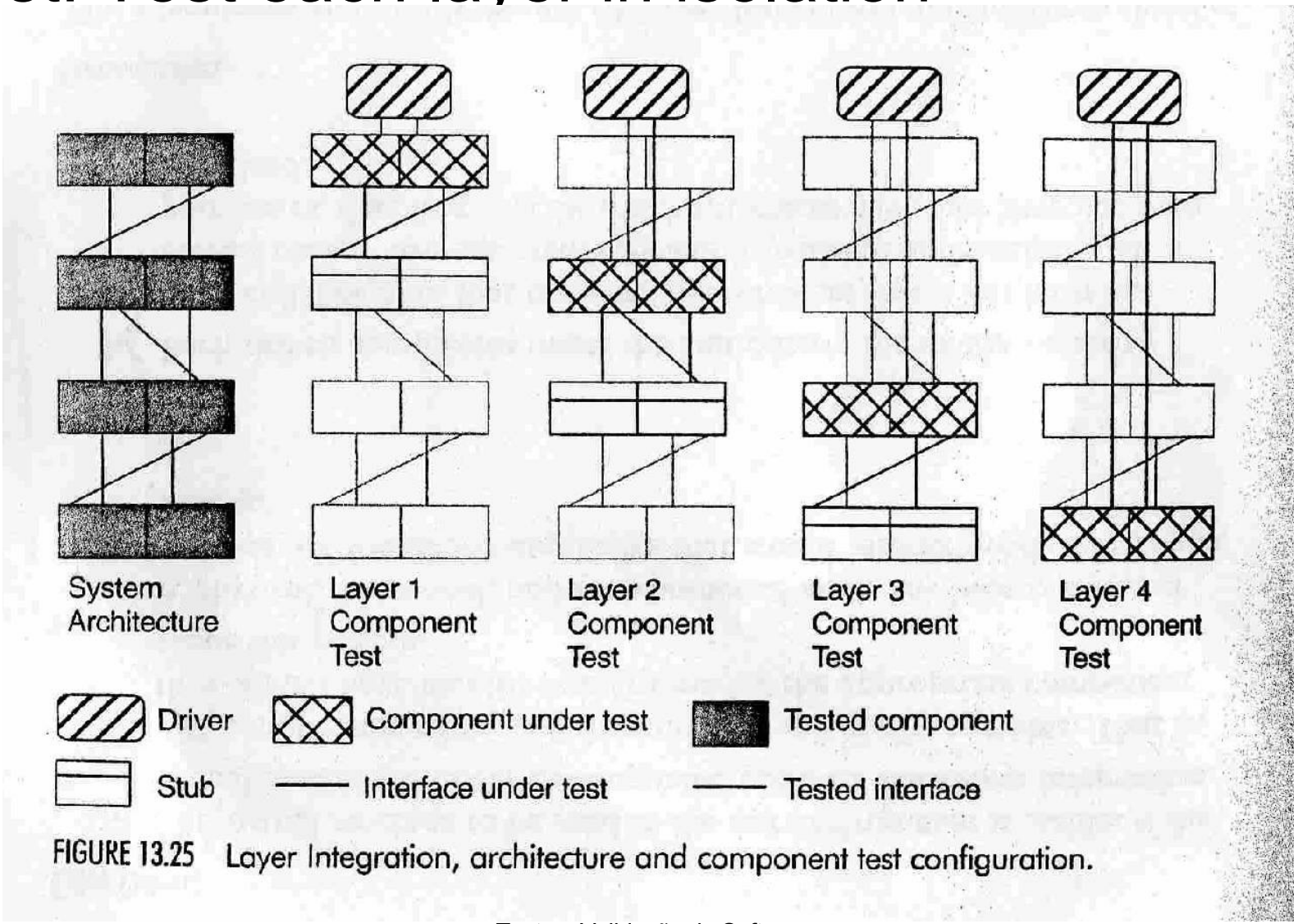
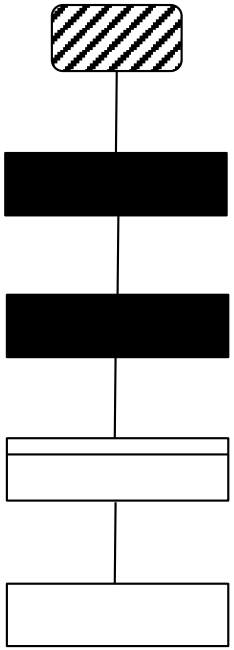


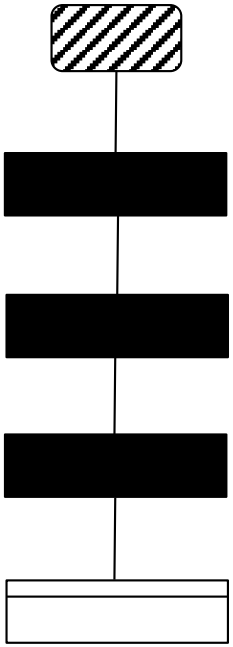
FIGURE 13.25 Layer Integration, architecture and component test configuration.

Strategy: Test Procedure 2nd part – Top-down case

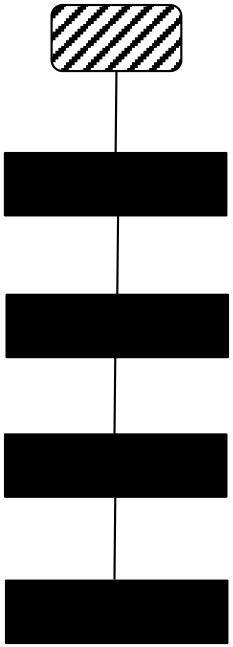
Layer 1-2 Integration



Layer 2-3 Integration



Layer 3-4 Integration



Strategy: Automation

- Requires drivers and stubs for each layer
- Top layer driver should be designed to exercise the entire system
  - If applying the top-down version

Entry and Exit Criteria

- Entry Criteria
  - Components of each collaboration are minimally operable

- Exit Criteria
  - Each driven component meets its exit criteria
  - Each collaboration traversing two or more layers has been exercised at least once
    - Verify with code coverage

Consequences

- Advantages and disadvantages are the same as for **Top-down Integration** (for the top-down variant) or **Bottom-up Integration** (for the bottom-up variant)

Client/Server Integration

- Intent
  - Demonstrate stability of client/server interaction
  - Begin by testing clients and servers in isolation and then use controlled increases in scope until all interfaces have been exercised
- Contex
  - Client/Server architecture
  - Concurrent development and testing possible
  - No single locus of control exists, each component of the system has its own control strategy
  - Achieves stepwise verification of interfaces between clients loosely coupled to servers

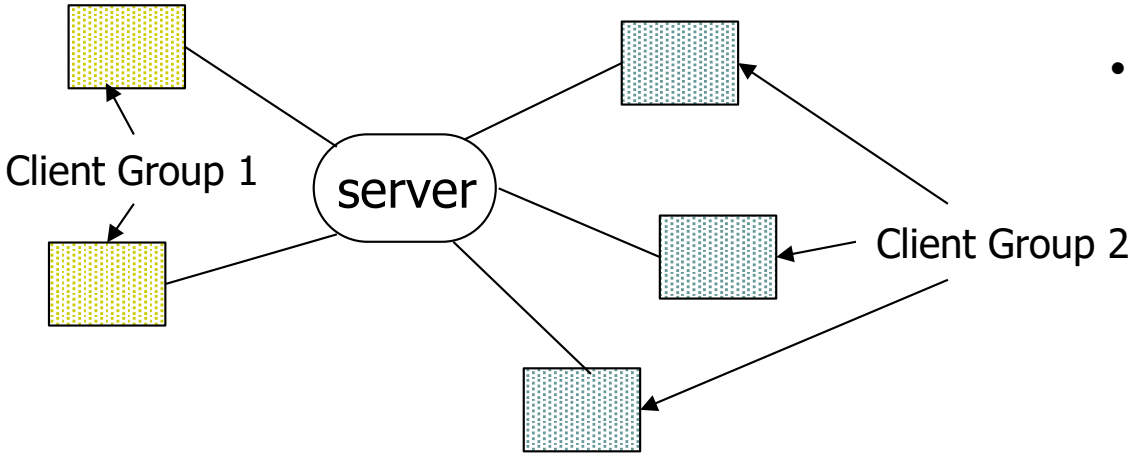
Strategy: Test model

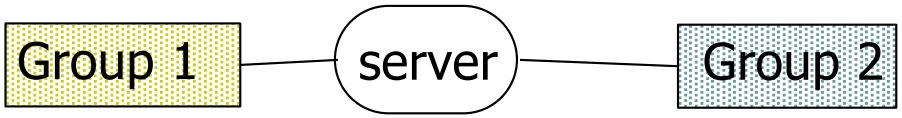
- Identify clients and servers
- Use **Extended Use Case Test** or **Round-trip Scenario Test** to model client/server interaction
- Exercise all combinations of clients and servers

Strategy: Test procedure

1. Each client is tested with a stub for the server
2. The server is tested with stubs for all client types
3. Test actual pairs of clients and servers
   - Server may retain stubs for other clients
   - Note: number of potential clients can be huge
4. Finally, all stubs are removed, and the individual use cases are replayed

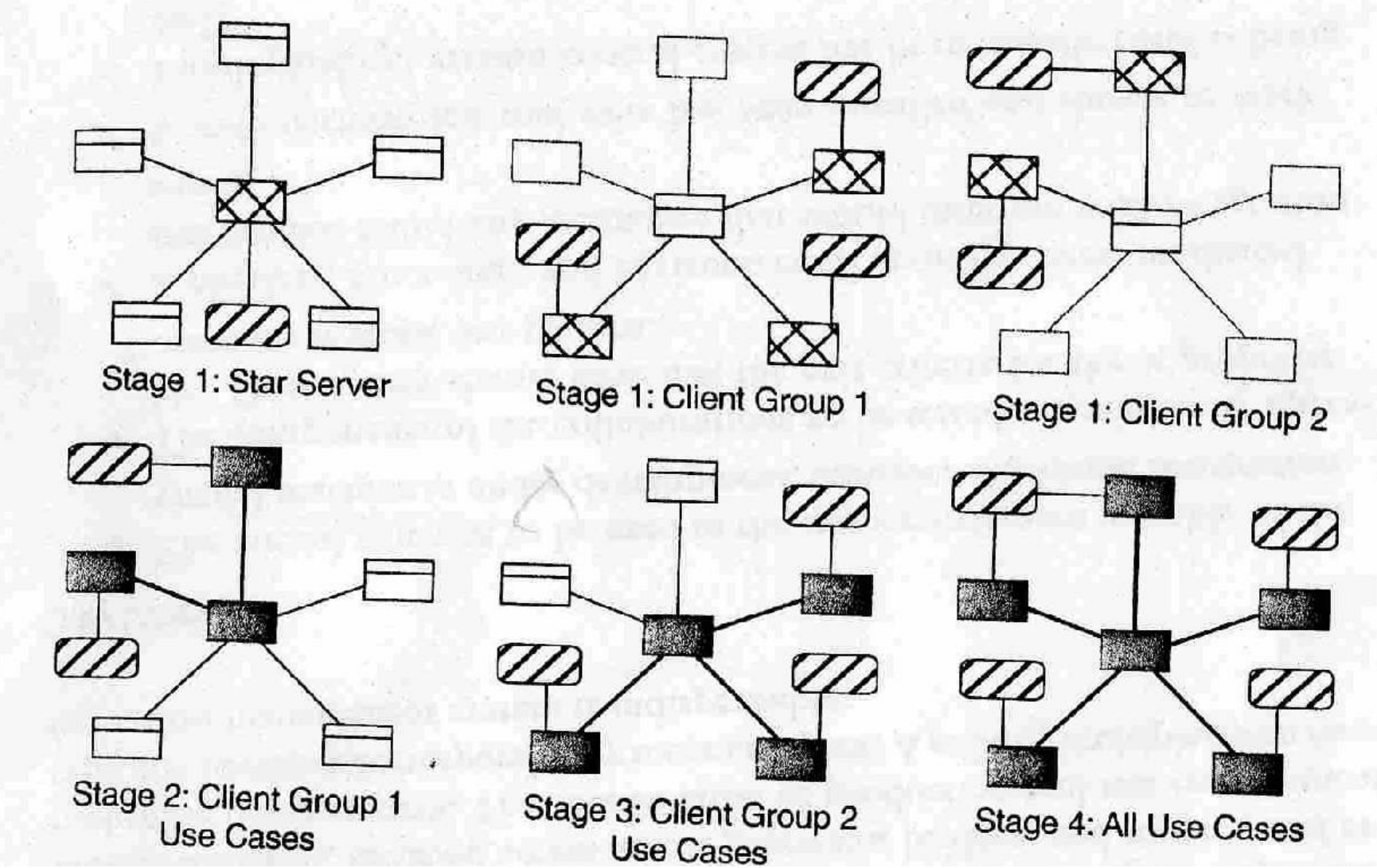
Strategy: Test procedure





- How to integrate every unique client in large systems?
  1. Identify client groups
  2. Consider group as representative client
  3. Test as having only the representative clients

Strategy: Test procedure



Entry and exit Criteria

- Entry Criteria
  - Components are minimally operable
  - Multiplatform test tool suite installed and working
- Exit Criteria
  - Each driven component meets exit criteria for its pattern
  - Interface to each subcomponent has been exercised at least once
    - Distribution limits coverage tools usefulness
    - Some effort should still be made to establish coverage of each end-to-end path

Consequences

- Disadvantages:
  - Cost of driver and stub development for clients and servers
  - Cannot exercise end-to-end use cases until midway or late in the testing cycle
- Advantages:
  - Avoids big bang integration problems
  - Order can be sequenced according to priority or risk
    - Client and server integration order has few constraints
  - Approach supports controllable and repeatable testing

Collaboration Integration

- Intent
  - Demonstrate stability by adding sets of components to the SUT that are required to support a given collaboration
- Context
  - Exercises interfaces between participants of a collaboration and organizes integration according to collaborations
  - Integration is complete when all components and interfaces have been tested, which may not require testing every collaboration
  - Use when:
    - The system has clearly defined collaborations that cover all components
    - Demonstrating a working collaboration early is important
    - Credible demonstration requires more than just a pass through the control hierarchy

Backbone Integration

- Intent
  - Combines Top-down, Bottom-up and Big Bang Integration to verify interoperability among tightly coupled subsystems
- Context
  - Embedded system applications and their infrastructure are often developed at the same time
    - The application cannot operate without the backbone
    - The backbone provides services that are essential for running tests and the application
    - Stub the backbone would be impractical
    - The high-level control strategy can be exercised with stubs for the upper and middle levels
    - The middle level consists of several clusters that have loose intercluster coupling, but tight intracluster coupling

Distributed Services Integration

- Intent
  - Demonstrate stability of interaction among loosely coupled peer components
  - Begin by testing some nodes in isolation, then use controlled increases in scope until all interfaces have been exercised
- Context
  - Components run concurrently
  - No single locus of control
  - No single hierarchy of servers

High Frequency Integration

- Intent
  - Integrate new code with a stabilized baseline frequently to prevent integration bugs from going undiscovered and to prevent divergence from the working, stabilized baseline
- Context
  - Rapid incremental development can result in missing or conflicting capabilities
    - Frequent integration prevents the festering of this
  - Necessary conditions for beginning a High-frequency integration regime
    - A stable increment is available
    - Meaningful increments can be produced in the frequency interval
    - Test suite is developed in parallel with code and kept current
    - High Frequency Integration must be automated
    - Configuration management tool must be installed and is routinely used