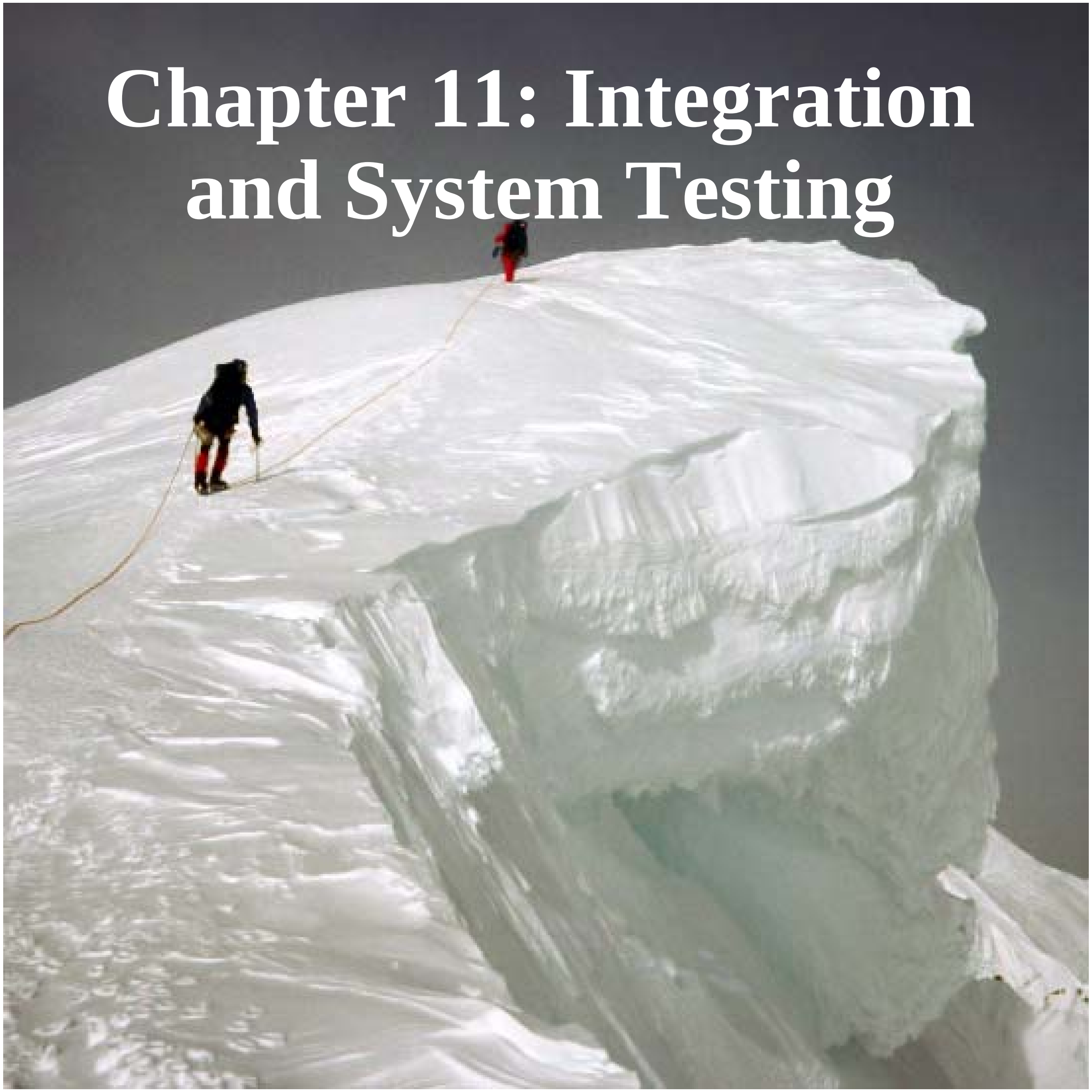
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 11: Integration and System Testing

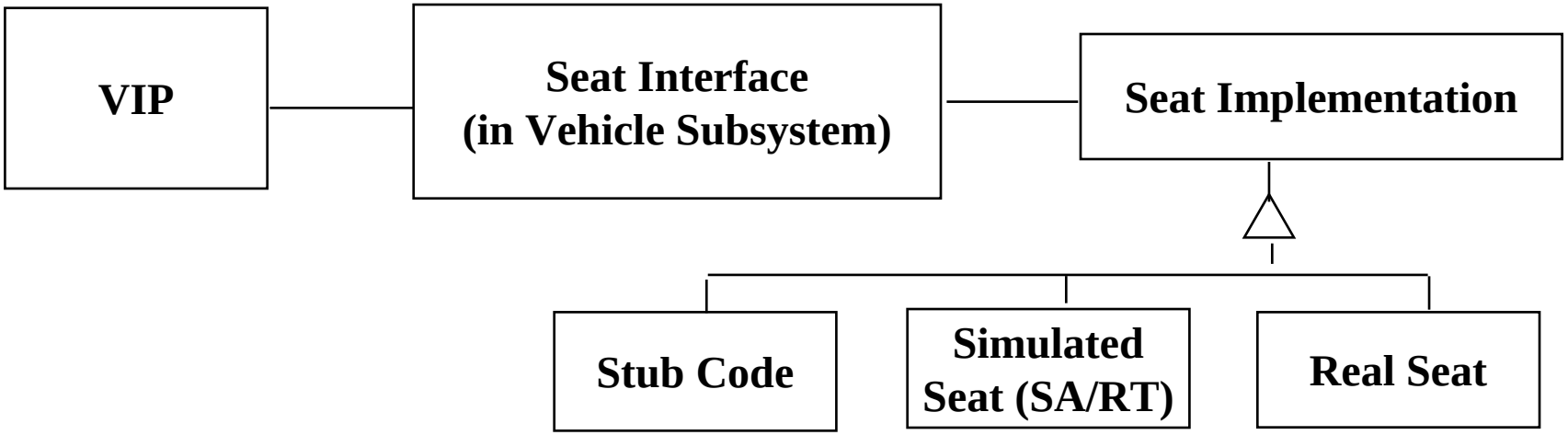


Integration Testing Strategy

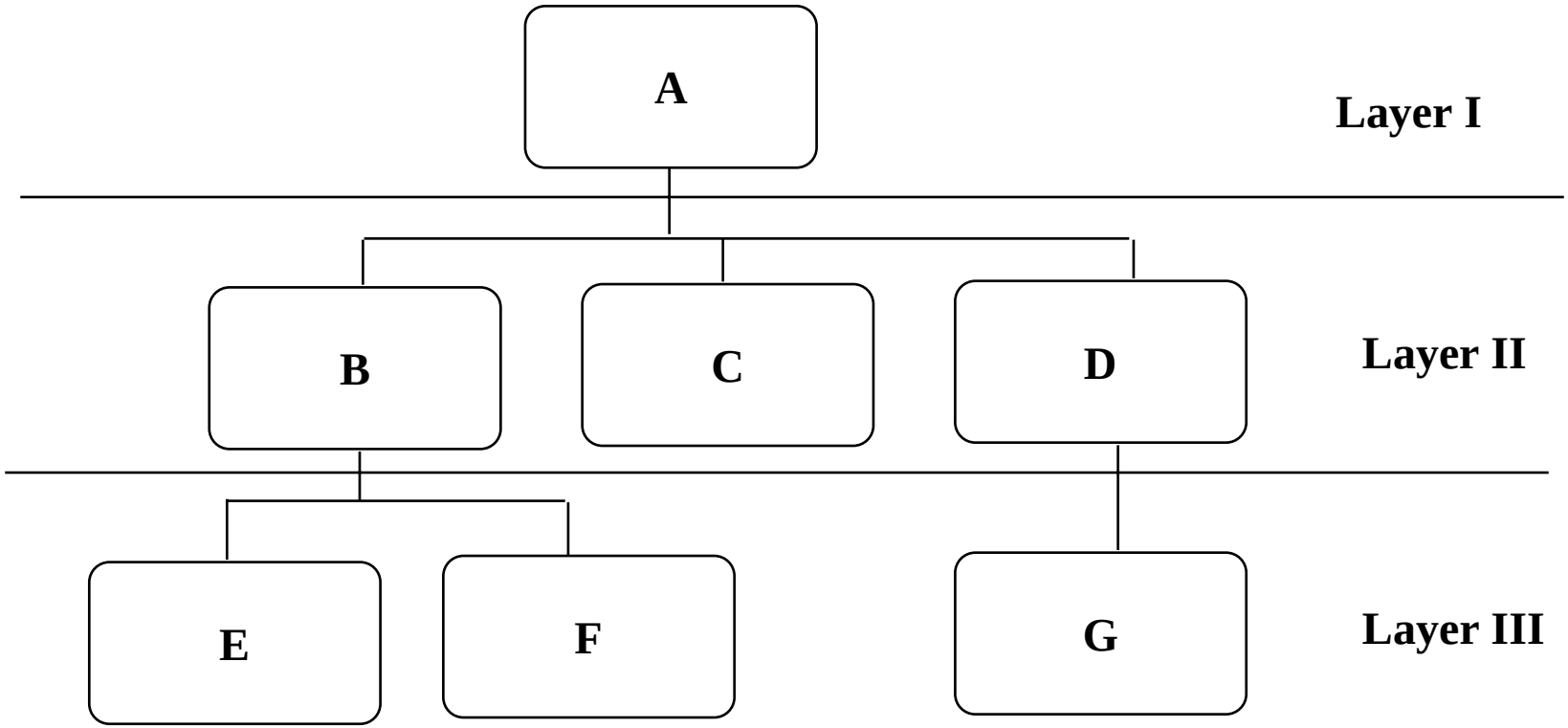
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - **Big bang integration (Nonincremental)**
  - **Bottom up integration**
  - **Top down integration**
  - **Sandwich testing**
  - **Variations of the above**
- For the selection use the system decomposition from the System Design

Using the Bridge Pattern to enable early Integration Testing

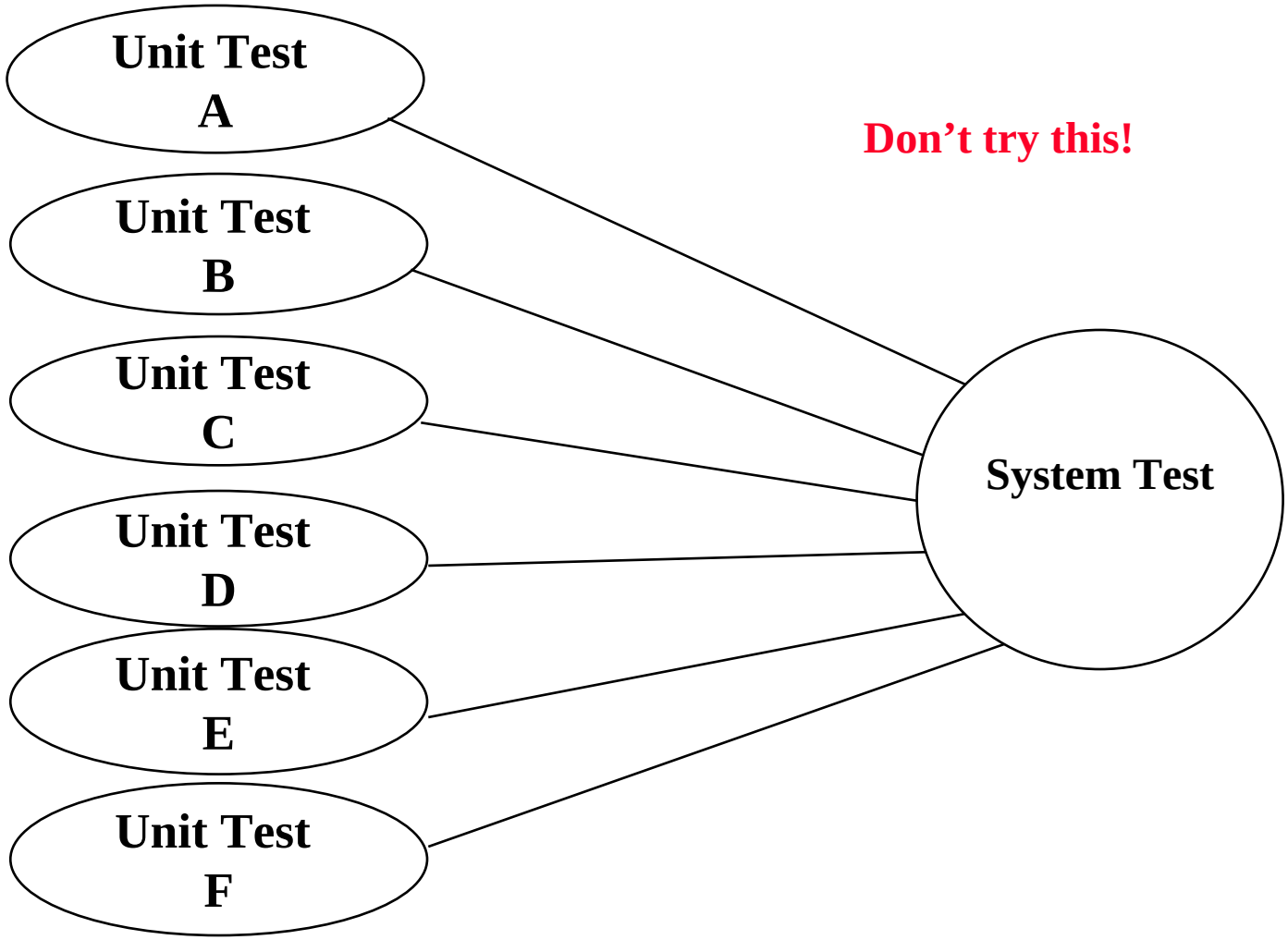
- Use the bridge pattern to provide multiple implementations under the same interface.
- Interface to a component that is incomplete, not yet known or unavailable during testing



Example: Three Layer Call Hierarchy

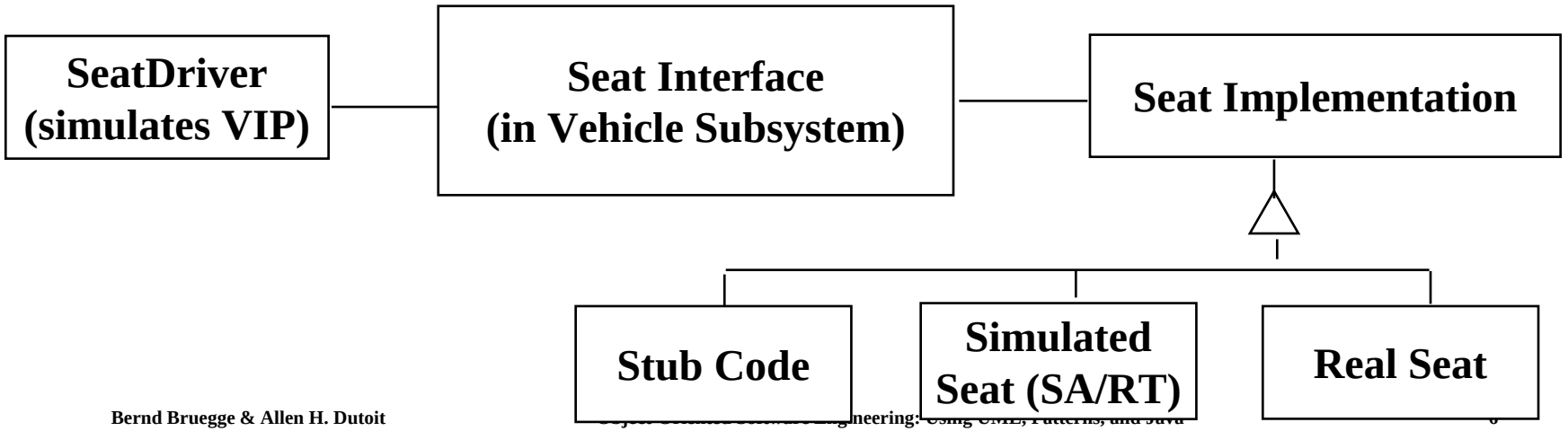


Integration Testing: Big-Bang Approach

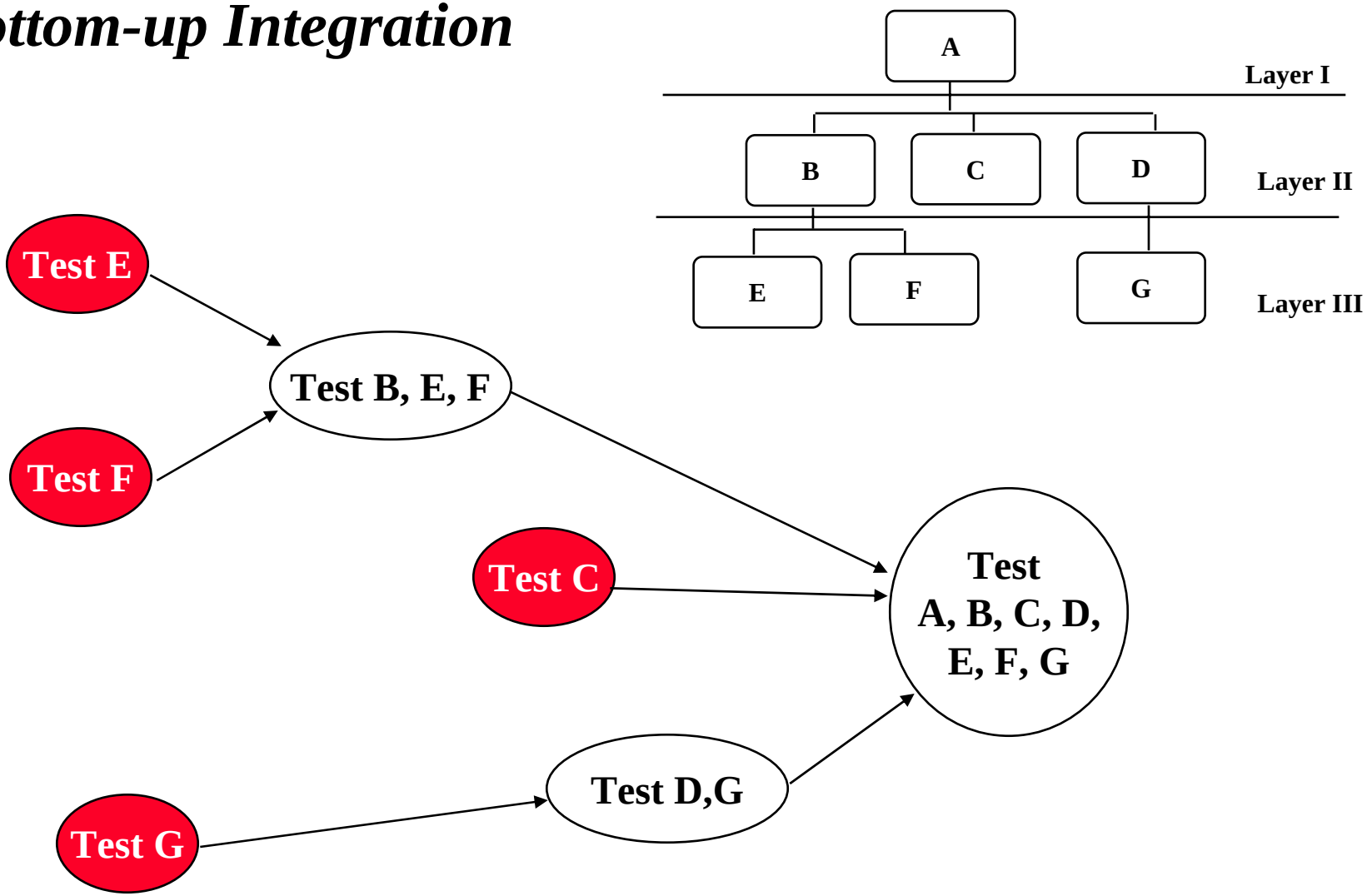


Bottom-up Testing Strategy

- The subsystem 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 done repeatedly until all subsystems are included in the testing
- Special program needed to do the testing, Test Driver:
  - **A routine that calls a subsystem and passes a test case to it**



Bottom-up Integration



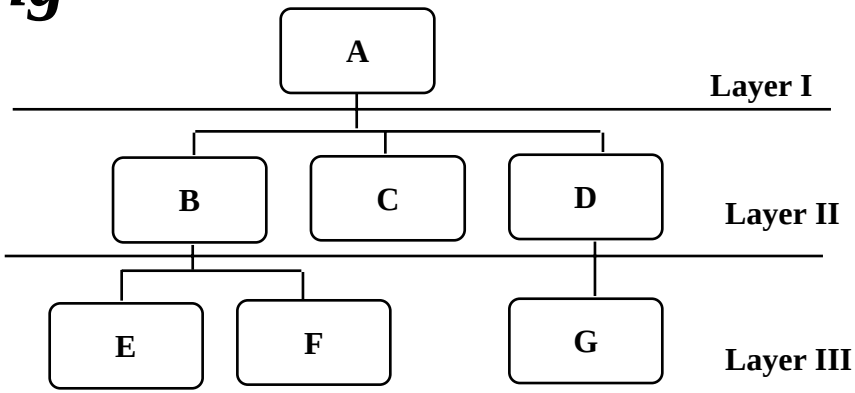
Pros and Cons of bottom up integration testing

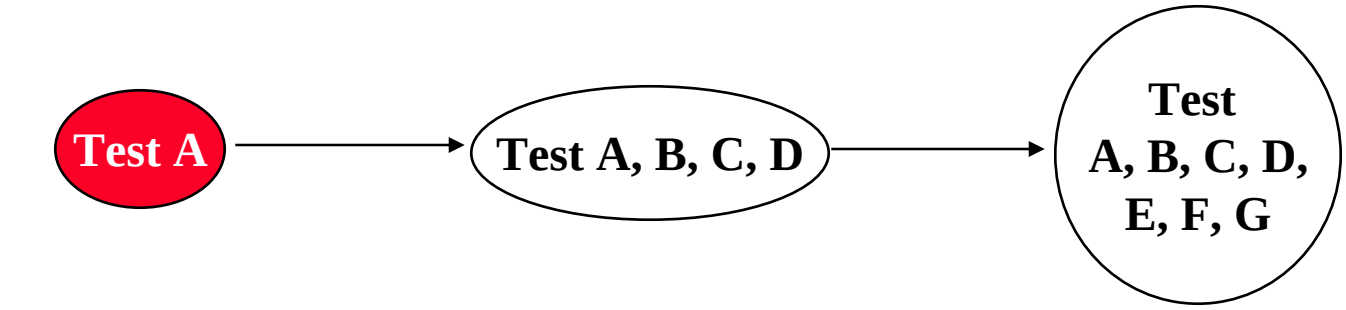
- Bad for functionally decomposed systems:
  - **Tests the most important subsystem (UI) last**
- Useful for integrating the following systems
  - **Object-oriented systems**
  - **real-time systems**
  - **systems with strict performance requirements**

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
- Special program is needed to do the testing, *Test stub :*
  - **A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.**

Top-down Integration Testing





Layer I

Layer I + II

All Layers

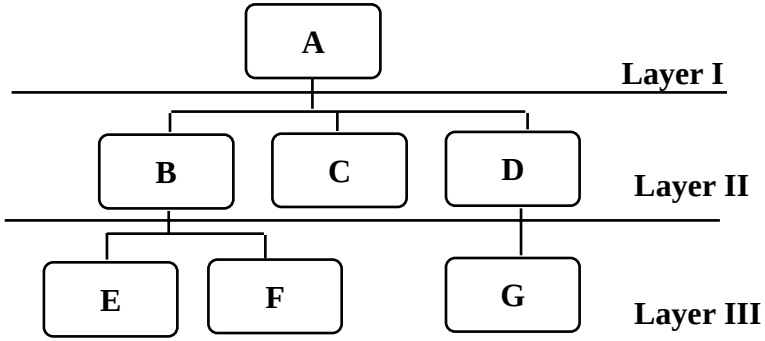
Pros and Cons of top-down integration testing

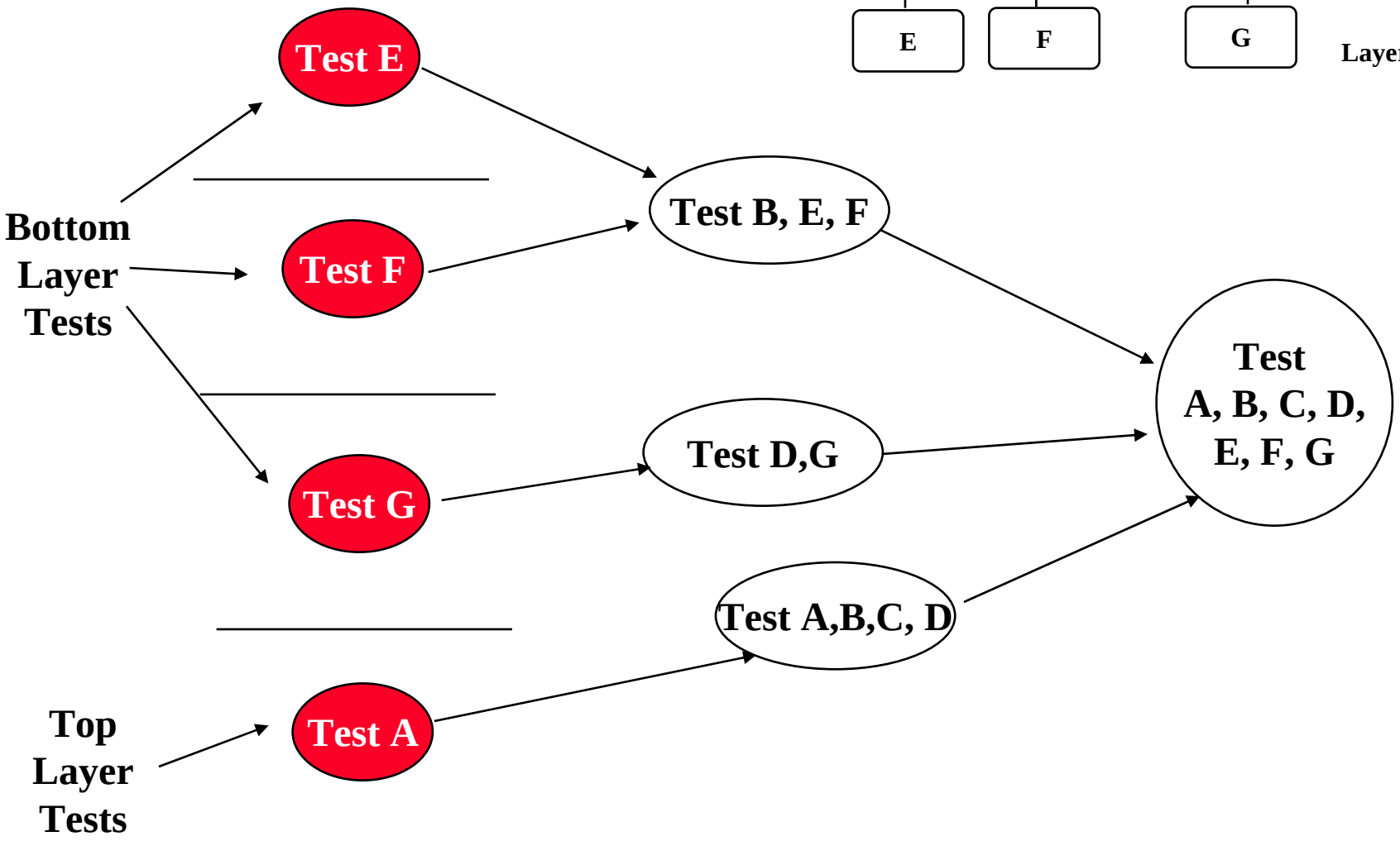
- Test cases can be defined in terms of the functionality of the system (functional requirements)
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
- One solution to avoid too many stubs: *Modified top-down testing strategy*
  - **Test each layer of the system decomposition individually before merging the layers**
  - **Disadvantage of modified top-down testing: Both, stubs and drivers are needed**

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is view 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**
- How do you select the target layer if there are more than 3 layers?
  - **Heuristic: Try to minimize the number of stubs and drivers**

Sandwich Testing Strategy





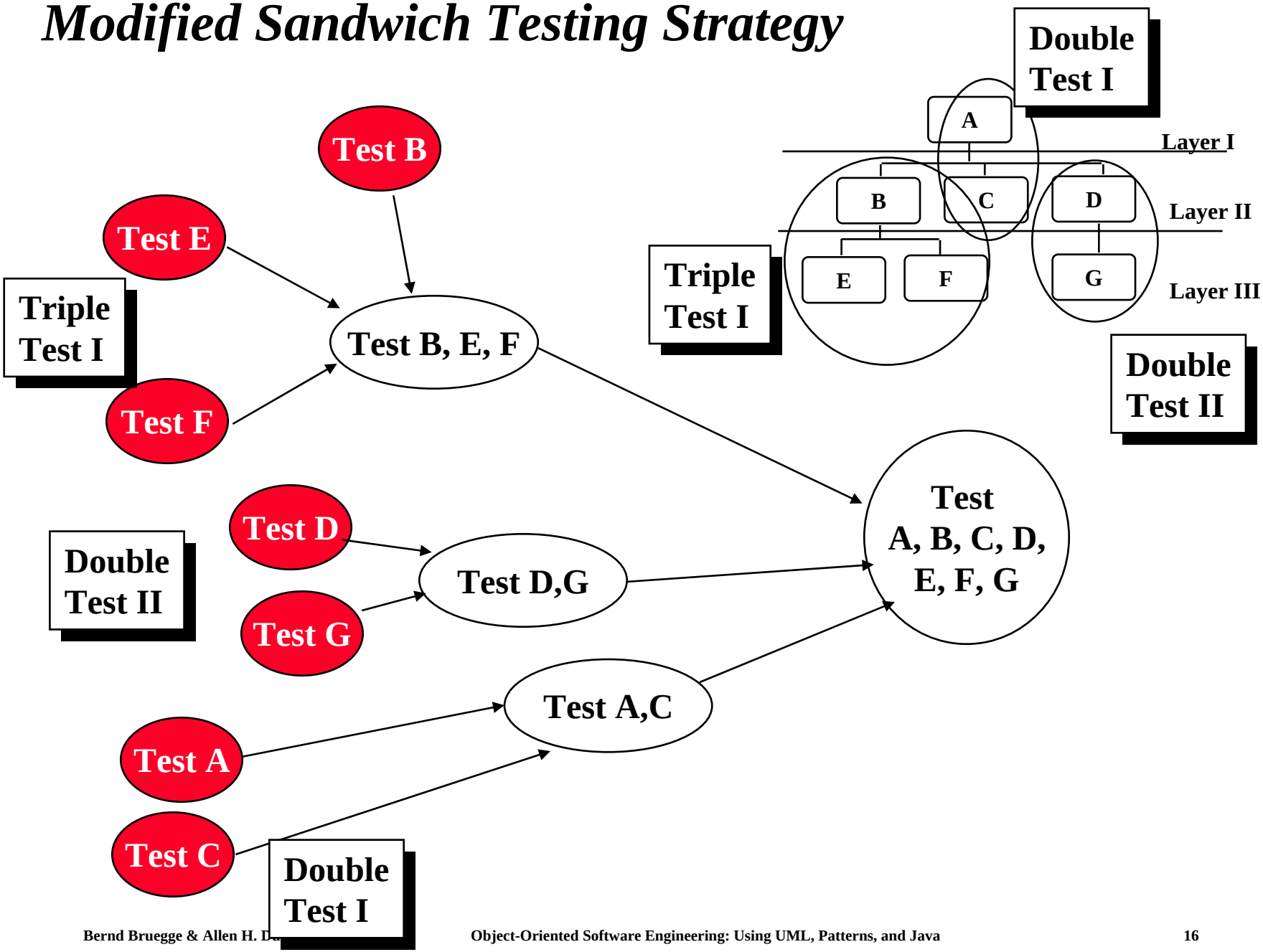
Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy

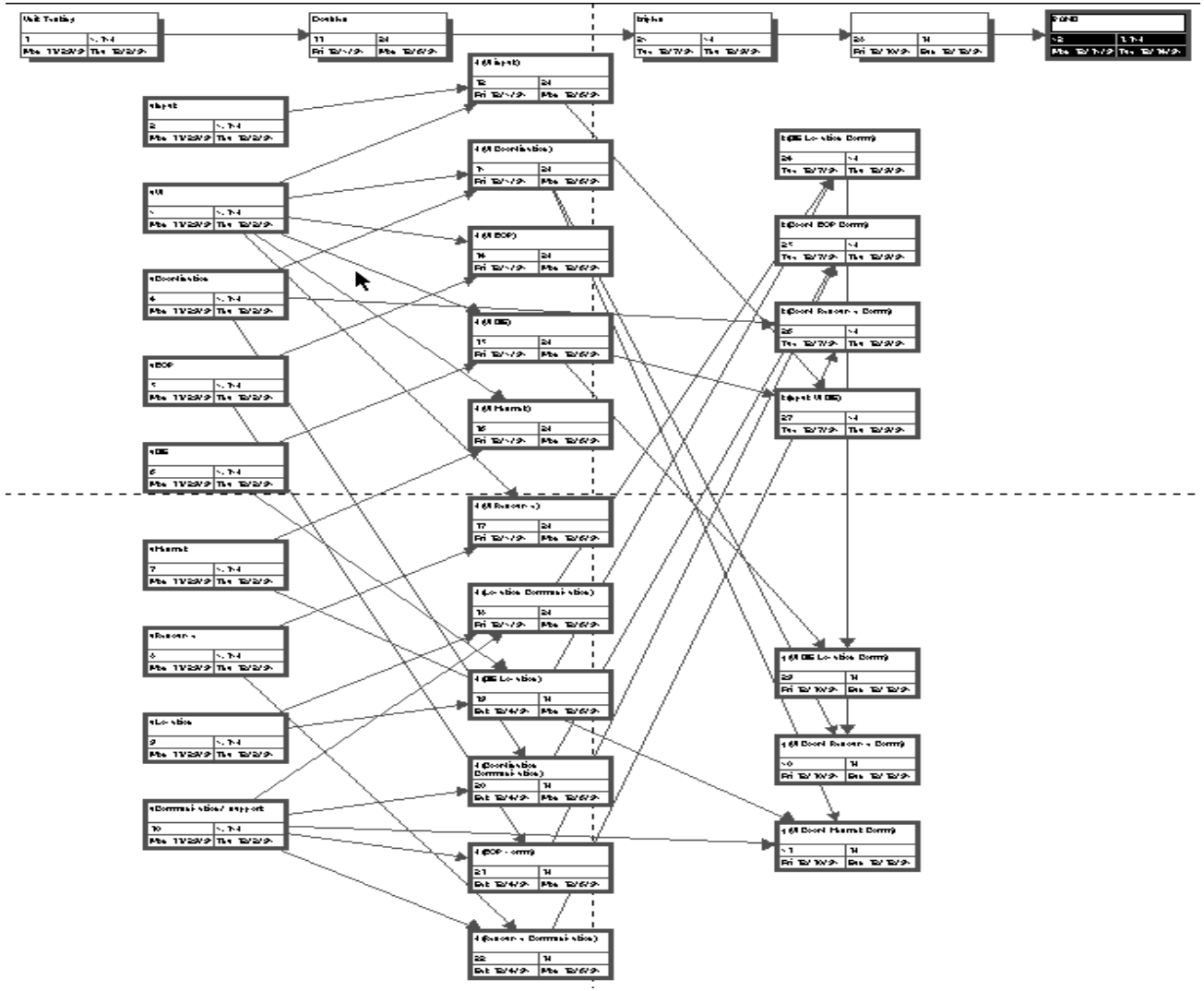
Modified Sandwich Testing Strategy

- 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)**

Modified Sandwich Testing Strategy



Scheduling Sandwich Tests: Example of a Dependency Chart



Unit Tests **Double Tests** **Triple Tests** **SystemTests**

Steps in Integration-Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing:* Define test cases that exercise all uses cases with the selected component
4. Do *structural testing:* Define test cases that exercise the selected component
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

Which Integration Strategy should you use?

- 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**
  - **Writing stubs can be difficult**

System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

Impact of requirements on system testing:

- **The more explicit the requirements, the easier they are to test.**
- **Quality of use cases determines the ease of functional testing**
- **Quality of subsystem decomposition determines the ease of structure testing**
- **Quality of nonfunctional requirements and constraints determines the ease of performance tests:**

Structure Testing

- *Essentially the same as white box testing.*
- Goal: Cover all paths in the system design
  - **Exercise all input and output parameters of each component.**
  - **Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)**
  - **Use conditional and iteration testing as in unit testing.**

Functional Testing

Essentially the same as black box testing

- Goal: Test functionality of system
- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box.
- Unit test cases can be reused, but in end user oriented new test cases have to be developed as well.

Performance Testing

- Stress Testing
  - **Stress limits of system (maximum # of users, peak demands, extended operation)**
- Volume testing
  - **Test what happens if large amounts of data are handled**
- Configuration testing
  - **Test the various software and hardware configurations**
- Compatibility test
  - **Test backward compatibility with existing systems**
- Security testing
  - **Try to violate security requirements**
- Timing testing
  - **Evaluate response times and time to perform a function**
- Environmental test
  - **Test tolerances for heat, humidity, motion, portability**
- Quality testing
  - **Test reliability, maintain- ability & availability of the system**
- Recovery testing
  - **Tests system's response to presence of errors or loss of data.**
- Human factors testing
  - **Tests user interface with user**

Test Cases for Performance Testing

- **Push the (integrated) system to its limits.**
- **Goal: Try to break the subsystem**
- **Test how the system behaves when overloaded.**
  - **Can bottlenecks be identified? (First candidates for redesign in the next iteration**
- **Try unusual orders of execution**
  - **Call a receive() before send()**
- **Check the system's response to large volumes of data**
  - **If the system is supposed to handle 1000 items, try it with 1001 items.**
- **What is the amount of time spent in different use cases?**
  - **Are typical cases executed in a timely fashion?**

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - **Choice of tests is made by client/sponsor**
  - **Many tests can be taken from integration testing**
  - **Acceptance test is performed by the client, not by the developer.**
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

- *Alpha test:*
  - **Sponsor uses the software at the *developer's site.***
  - **Software used in a controlled setting, with the developer always ready to fix bugs.**
- *Beta test:*
  - **Conducted at *sponsor's site* (developer is not present)**
  - **Software gets a realistic workout in target environment**
  - **Potential customer might get discouraged**

Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

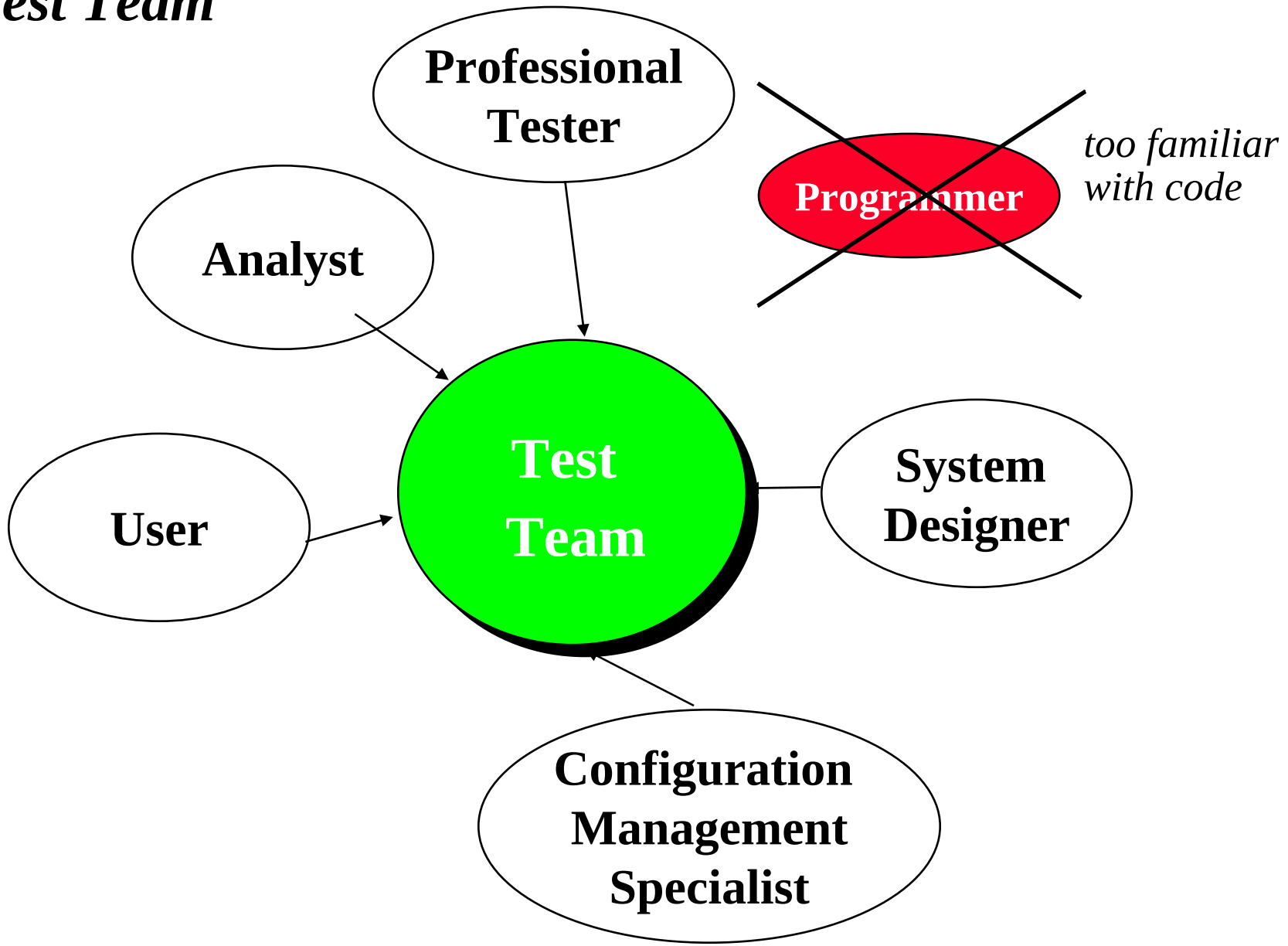
Execute the tests

Evaluate the test results

Change the system

Do regression testing

Test Team


Professional Tester
Programmer
too familiar with code
Analyst
Test Team
User
System Designer
Configuration Management Specialist

Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of component-testing (unit testing, integration testing) and system testing
- Design Patterns can be used for integration testing
- Testing has its own lifecycle