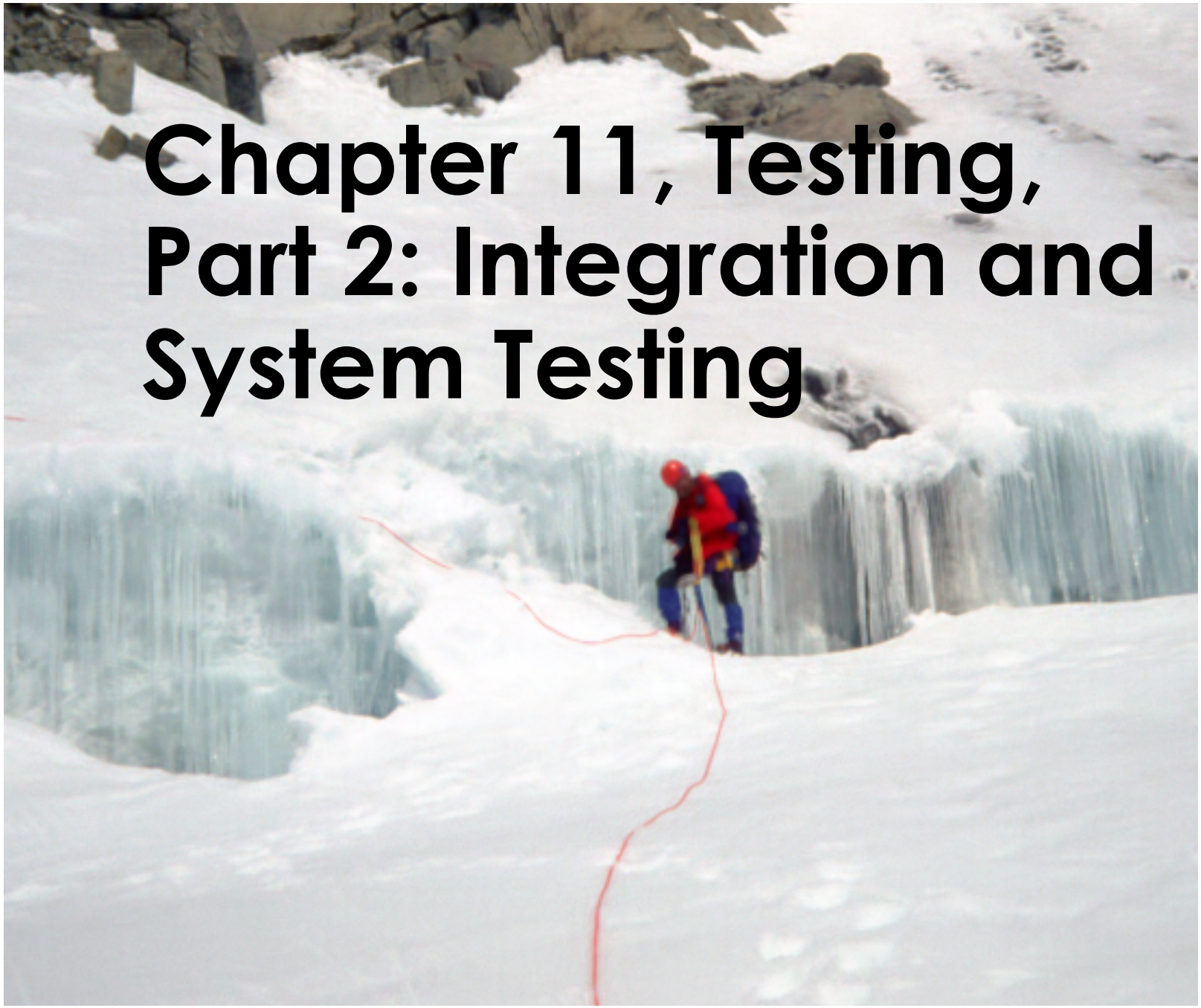
Object-Oriented Software Engineering
Using UML, Patterns, and Java



# Chapter 11, Testing, Part 2: Integration and System Testing

# Overview

- Integration testing
  - Big bang
  - Bottom up
  - Top down
  - Sandwich
- System testing
  - Functional
  - Performance
- Continuous Integration
- Acceptance testing
- Summary

# Integration Testing

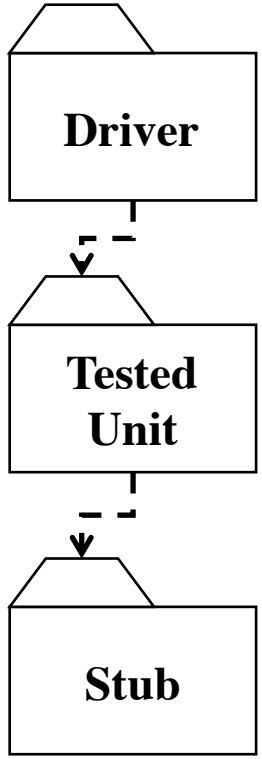
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The integration testing strategy determines the order in which the subsystems are selected for testing and integration.

# Why do we do integration testing?

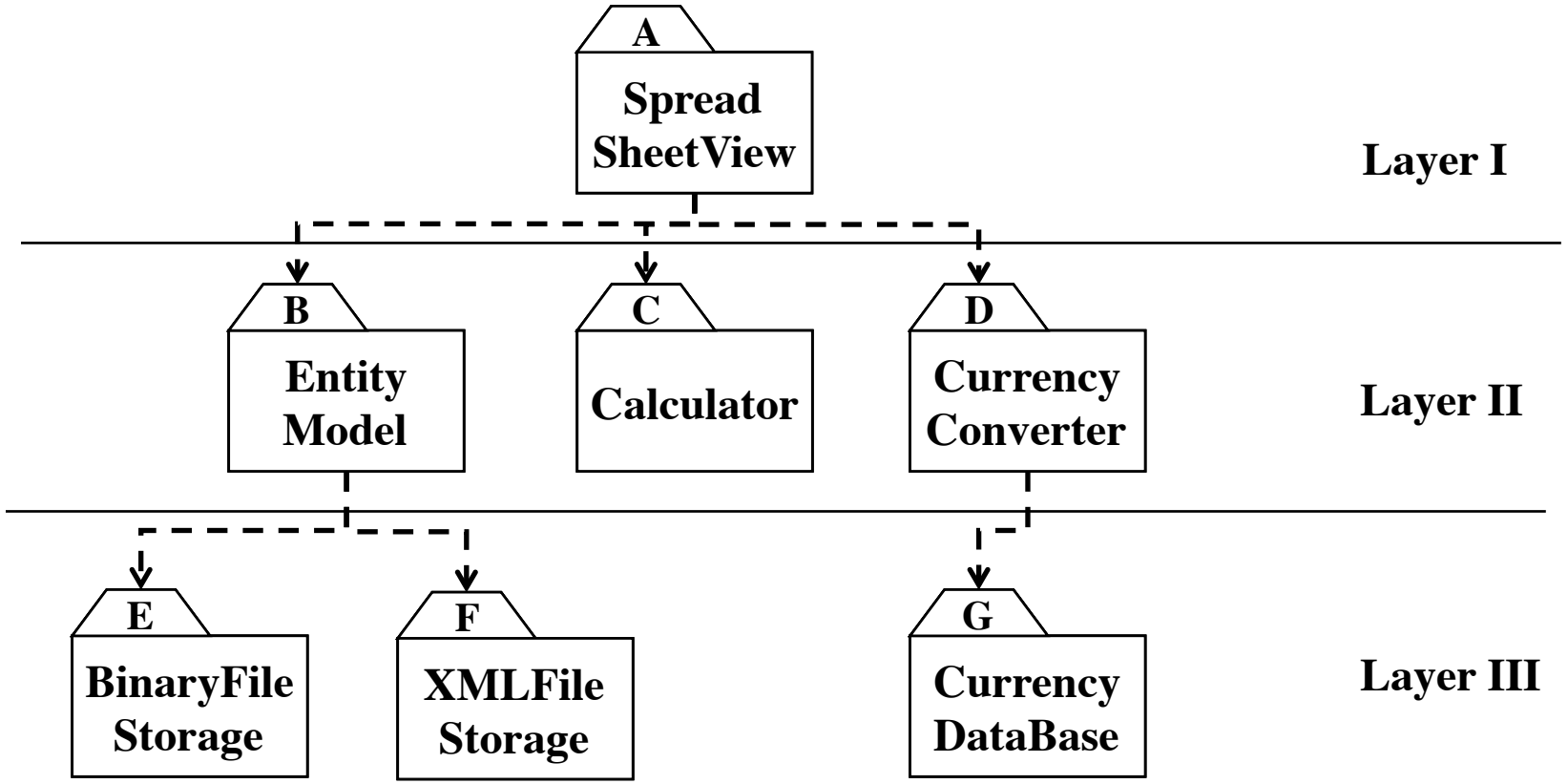
- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- When Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

# Stubs and drivers

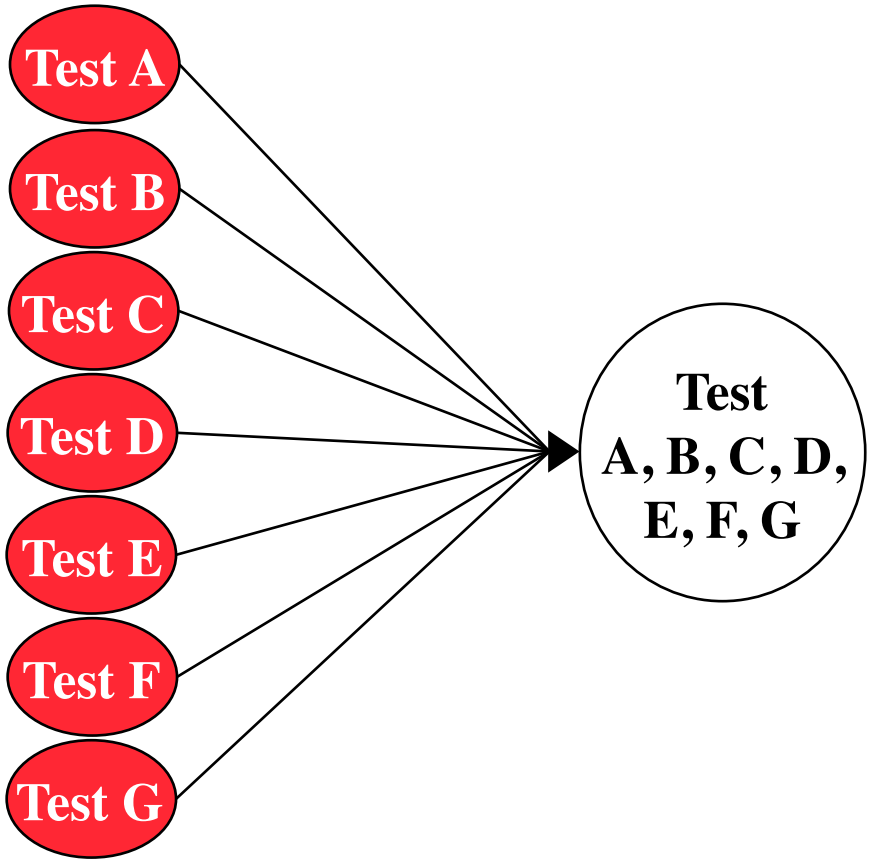
- Driver:
  - A component, that calls the `TestedUnit`
  - Controls the test cases
- Stub:
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.

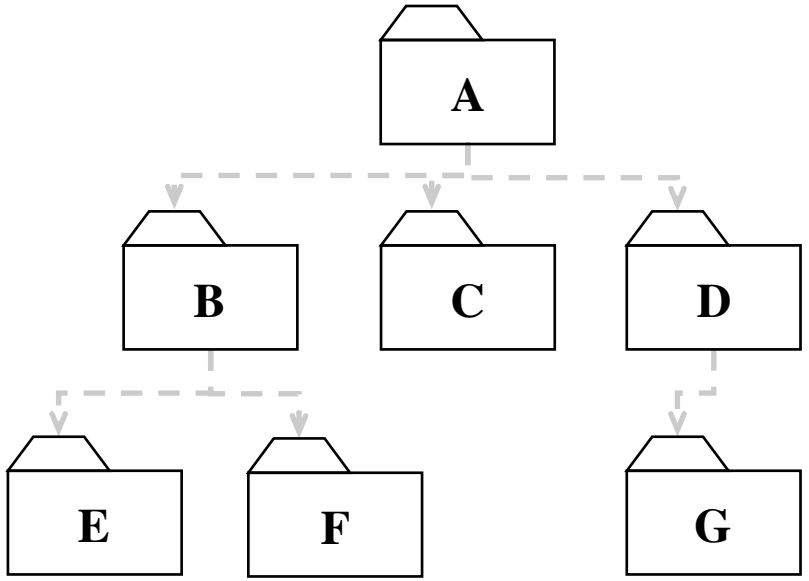


# Example: A 3-Layer-Design (Spreadsheet)



# Big-Bang Approach

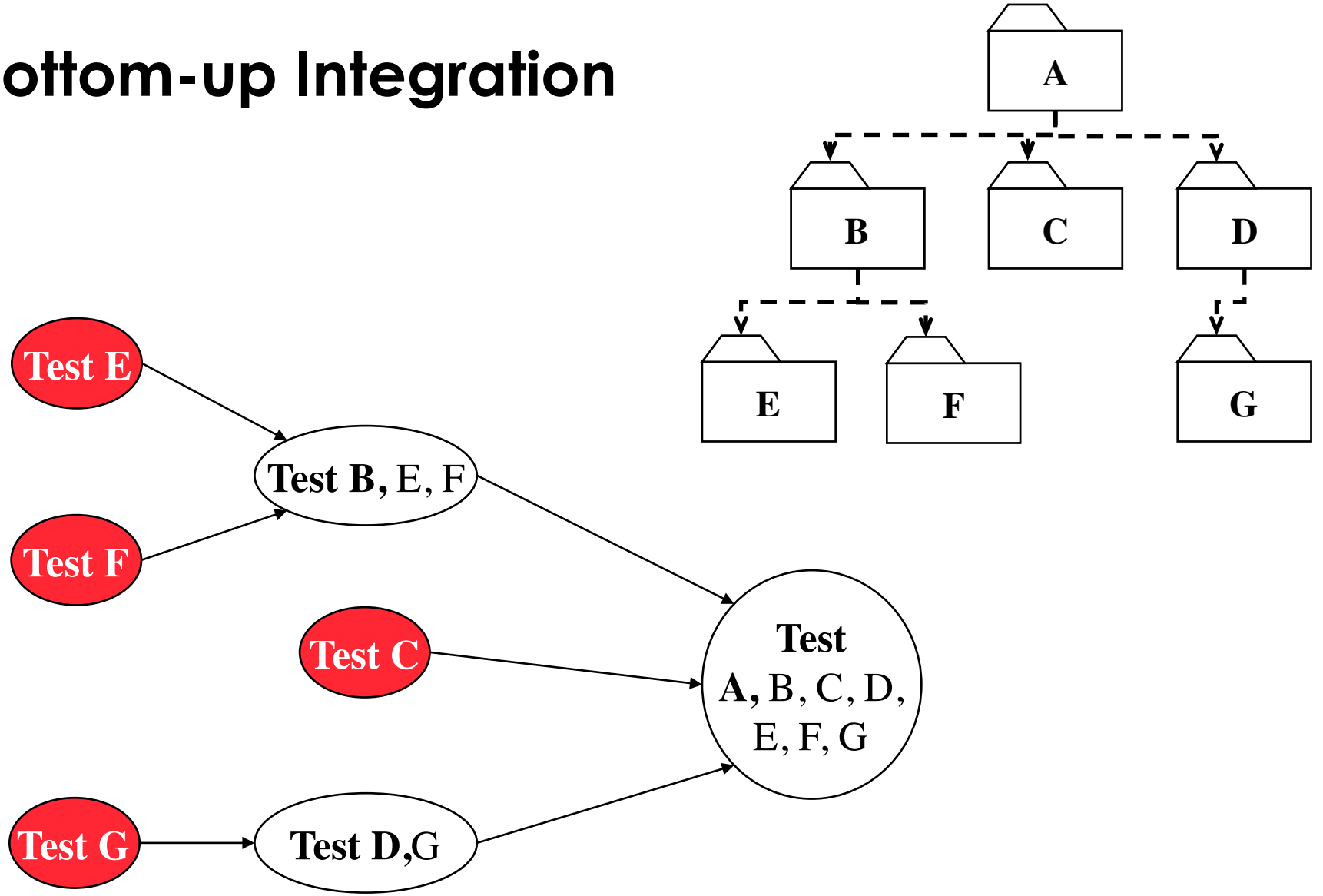


# Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the subsystems above this layer are tested that call the previously tested subsystems
- This is repeated until all subsystems are included.

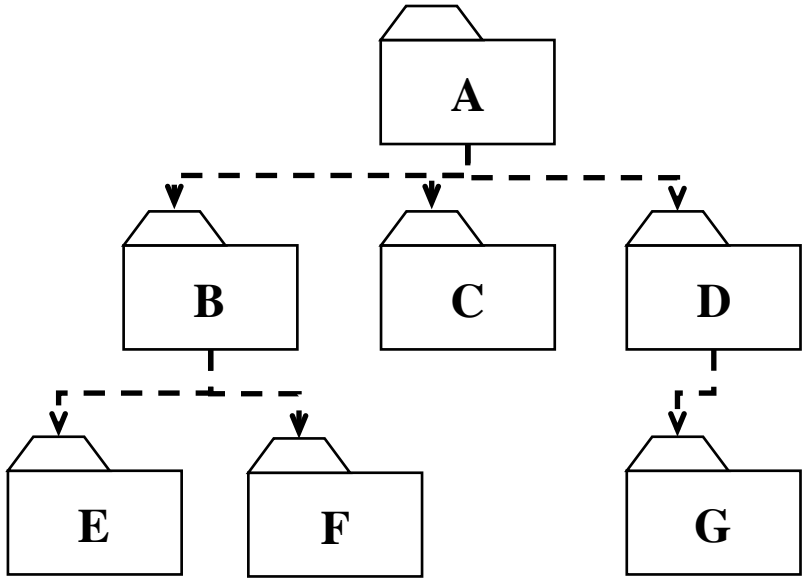
# Bottom-up Integration

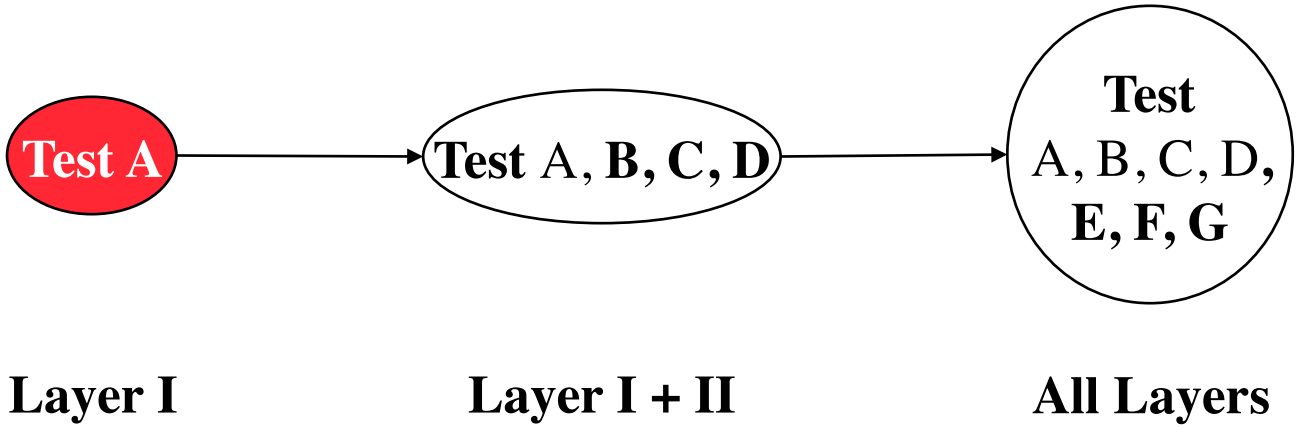


# Top-down Testing Strategy

- Test the subsystems in the top layer 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 tests.

# Top-down Integration

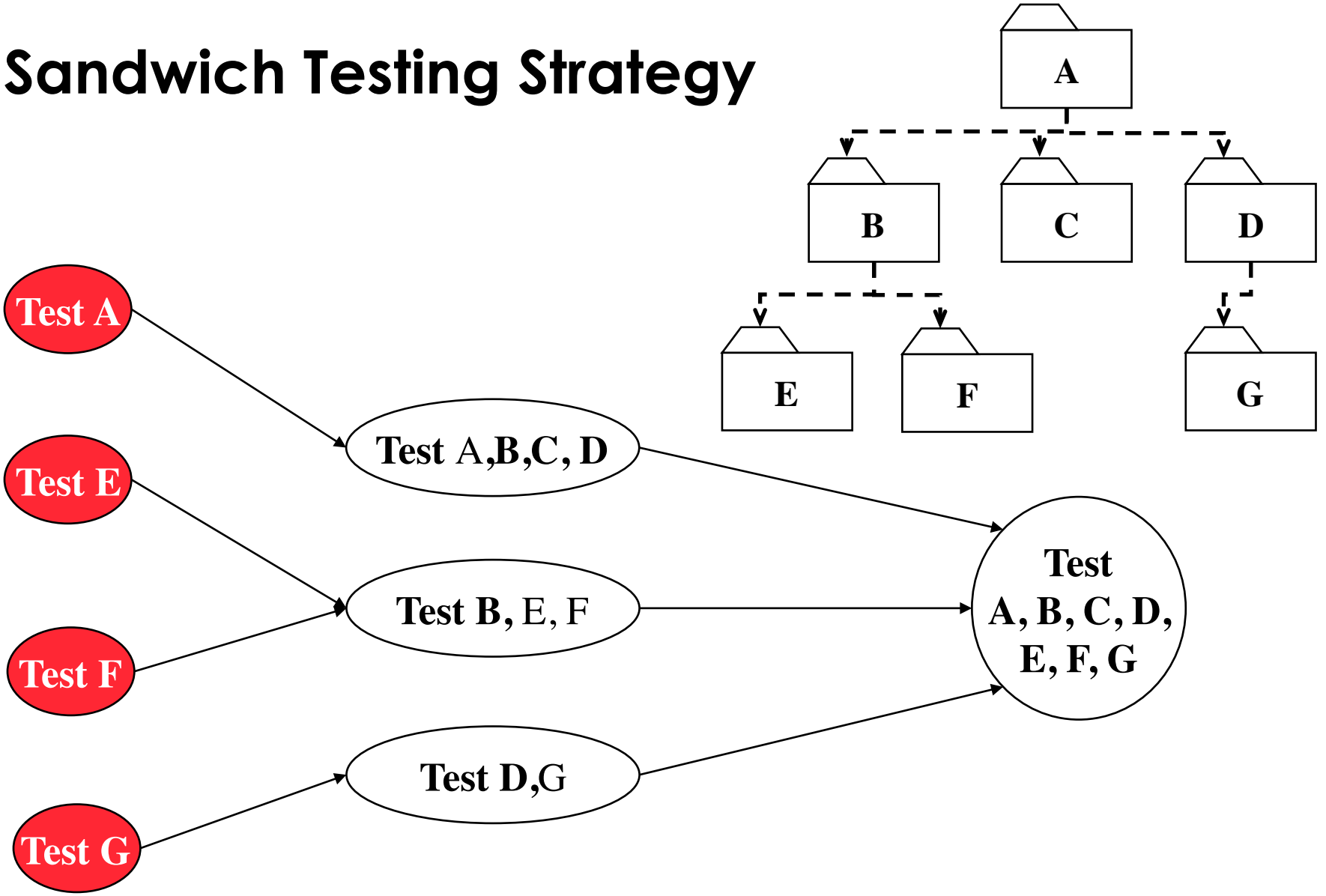


# Sandwich Testing Strategy

- 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.

# Sandwich Testing Strategy



# Pros and Cons: Top-Down Integration Testing

Pros:

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

Cons:

- Stubs are needed
- Writing stubs is difficult: Stubs must allow all possible conditions to be tested
- Large number of stubs may be required, especially if the lowest level of the system contains many methods
- Some interfaces are not tested separately.

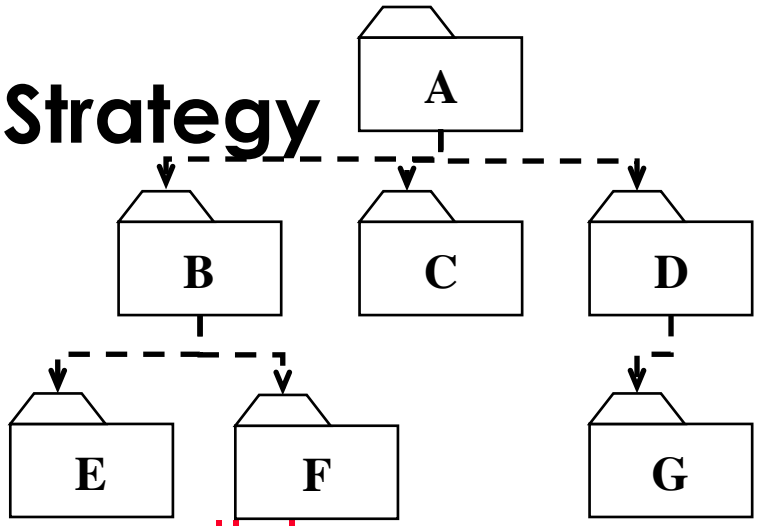
# Pros and Cons: Bottom-Up Integration Testing

- Pro
  - No stubs needed
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements
- Con:
  - Tests an important subsystem (the user interface) last
  - Drivers are needed.

# Pros and Cons of Sandwich Testing

- Pro:
  - Top and bottom layer tests can be done in parallel
- Con:
  - Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy.

# Modified Sandwich Testing Strategy



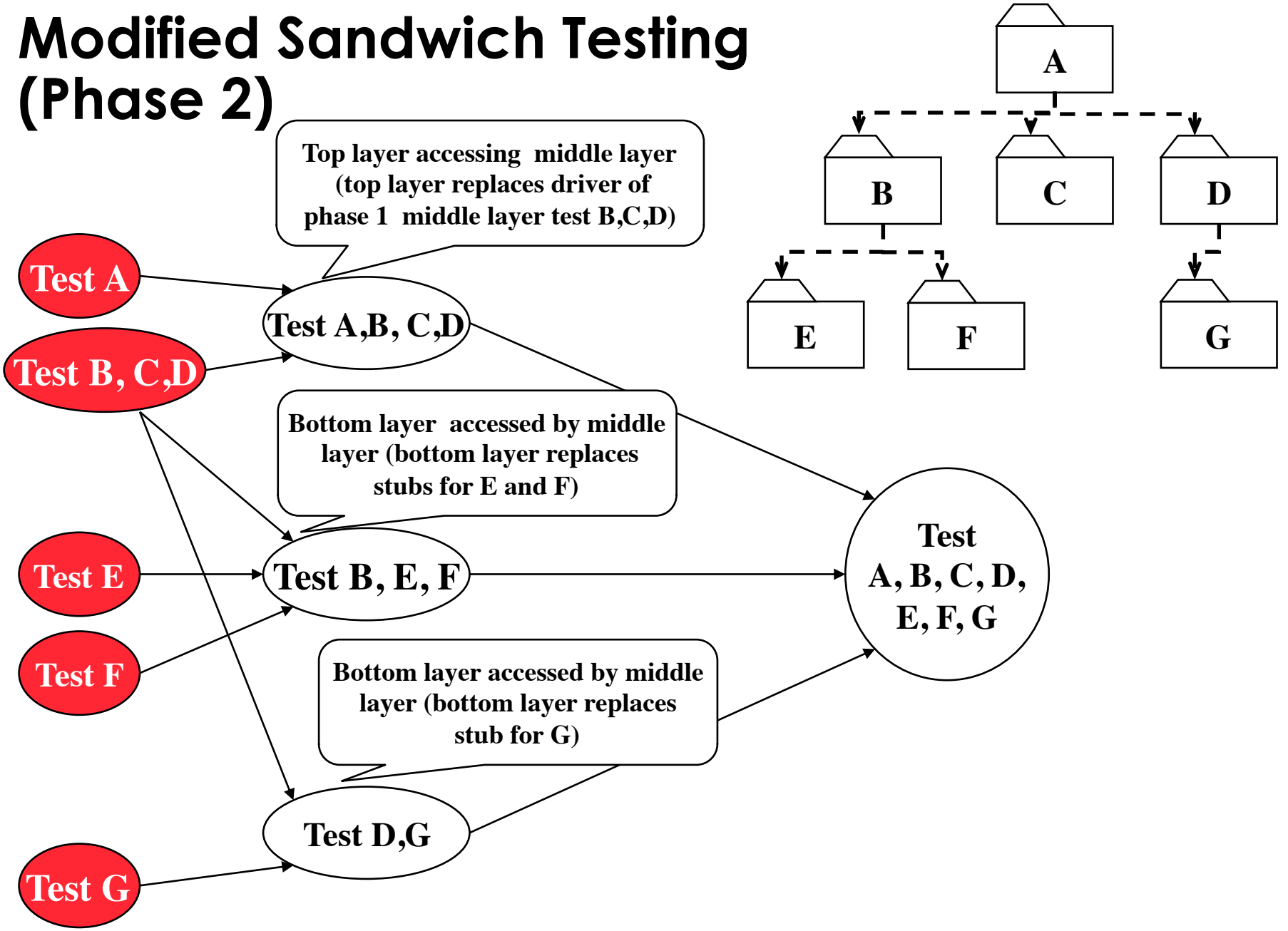

Phase 1: Three integration tests in parallel

- Top layer test with stubs for lower layers
- Middle layer test with drivers and stubs
- Bottom layer test with drivers for upper layers

Phase 2: Two more integration tests in parallel

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

# Modified Sandwich Testing (Phase 2)



# Risks in Integration Testing Strategies

- Risk #1: The higher the complexity of the software system, the more difficult is the integration of its components
- Risk #2: The later integration occurs in a project, the bigger is the risk that unexpected faults occur
- Bottom up, top down, sandwich testing (Horizontal integration strategies) don't do well with risk #2
- Continous integration addresses these risks by building as early and frequently as possible
- Additional advantages:
  - There is always an executable version of the system
  - Team members have a good overview of the project status.

# Definition Continuous Integration

Continuous Integration: A software development technique where members of a team *integrate* their work *frequently*, usually each person integrates at least daily, leading to multiple integrations per day.

Each integration is verified by an *automated build which includes the execution of tests - regres* to detect integration errors as quickly as possible.

Source: http://martinfowler.com/articles/continuousIntegration.html

# 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. Test functional requirements*:* Define test cases that exercise all uses cases with the selected component
4. Test subsystem decomposition*:* Define test cases that exercise all dependencies
5. Test non-functional requirements: 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 failures* with the (current) component *configuration*.

# System Testing

- Functional Testing
  - Validates functional requirements
- Performance Testing
  - Validates non-functional requirements
- Acceptance Testing
  - Validates clients expectations

# Functional 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 new test cases have to be developed as well.

# Performance Testing

Goal: Try to violate non-functional requirements

- 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?

# Types of Performance Testing

- Stress Testing
  - Stress limits of system
- 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
- Timing testing
  - Evaluate response times and time to perform a function
- Security testing
  - Try to violate security requirements
- Environmental test
  - Test tolerances for heat, humidity, motion
- Quality testing
  - Test reliability, maintain-ability & availability
- Recovery testing
  - Test system's response to presence of errors or loss of data
- Human factors testing
  - Test with end users.

# Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Alpha test:
  - Client uses the software at the developer's environment.
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- Beta test:
  - Conducted at client's environment (developer is not present)
  - Software gets a realistic workout in target environment

# Testing has many activities

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

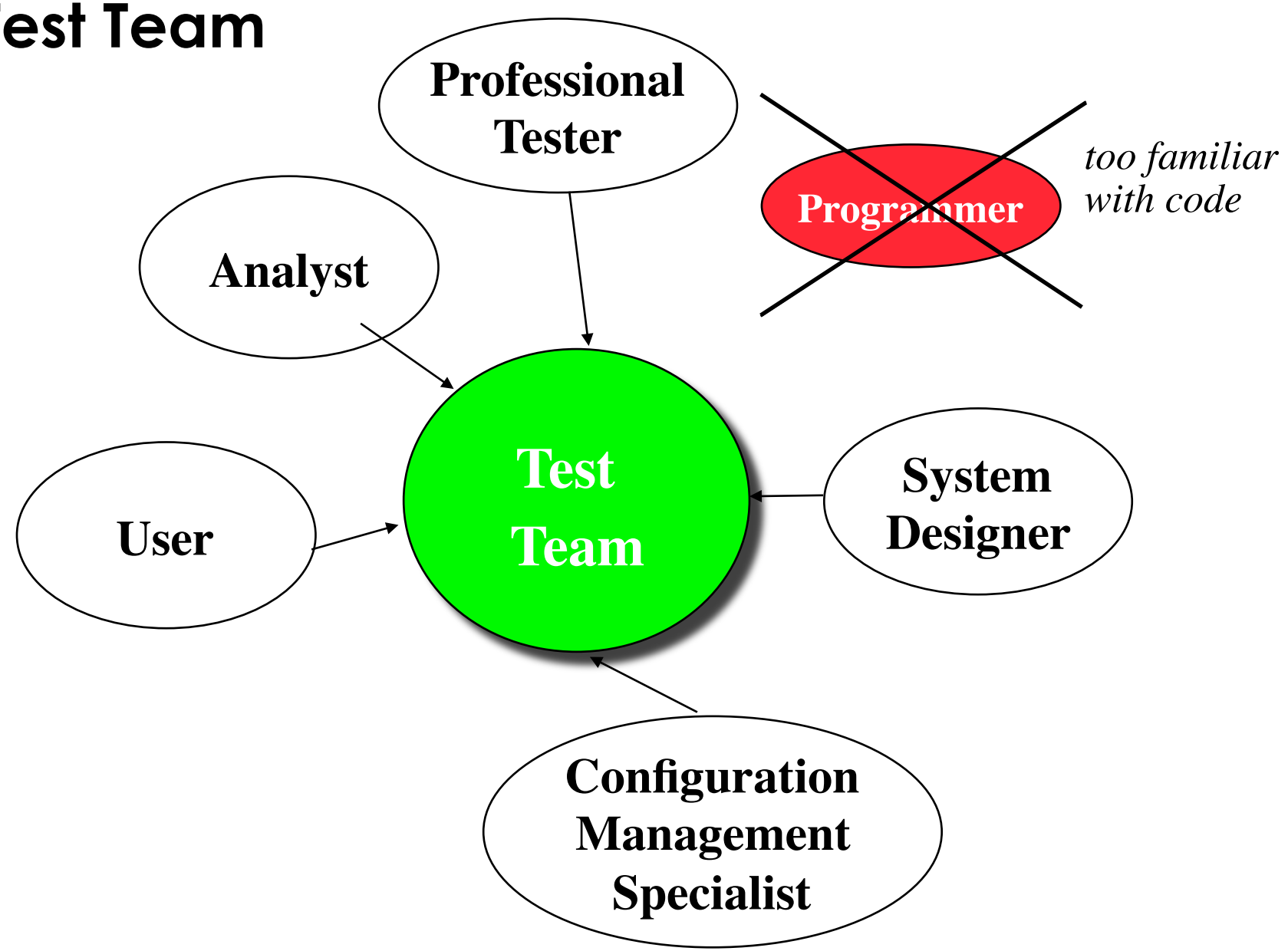
Execute the tests

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

# The 4 Testing Steps

1. Select what has to be tested
   - Analysis: Completeness of requirements
   - Design: Cohesion
   - Implementation: Source code
2. Decide how the testing is done
   - Review or code inspection
   - Proofs (Design by Contract, formal techniques)
   - Black-box, white box,
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)

3. Develop test cases
   - A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured
4. Create the test oracle
   - An oracle contains the predicted results for a set of test cases
   - The test oracle has to be written down before the actual testing takes place.

# Guidance for Test Case Selection

- Use *analysis knowledge* about functional requirements (black-box testing):
  - Use cases
  - Expected input data
  - Invalid input data
- Use *design knowledge* about system structure, algorithms, data structures (white-box testing):
  - Control structures
    - Test branches, loops, ...
  - Data structures
    - Test records fields, arrays, ...

- Use *implementation knowledge* about algorithms and datastructures:
  - Force a division by zero
  - If the upper bound of an array is 10, then use 11 as index.

# Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of
  - Unit testing
  - Integration testing
  - System testing
    - Acceptance testing
- Testing has its own lifecycle
- Recommended practice: Continous integration
  - Allows frequent integration *during* development (instead of *after* development).

# Additional Reading

- Martin Fowler, Continous Integration, 2006
  - http://martinfowler.com/articles/continuousIntegration.html
- Paul M. Duvall, Steve Matyas and Andrew Glover

  Continuous Integration: Improving Software Quality and Reducing Risk, Addison Wesley 2007
- Frameworks for Continous Integration
  - CruiseControl (Open Source)
    - http://cruisecontrol.sourceforge.net/
  - Hudson from Kohsuke Kawaguchi (Free Software)
    - http://weblogs.java.net/blog/kohsuke/archive/2009/08/announcing_sun.html