Software Engineering I: Software Technology

WS 2008/09

Integration Testing and System Testing

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Overview

- Integration testing techniques
  - Big bang (not a technique:-)
  - Bottom up testing
  - Top down testing
  - Sandwich testing
  - Continuous integration
- System testing
  - Functional testing
  - Performance testing
- Acceptance testing
- Summary

Odds and Ends

- This is the last lecture for this year
  - No lecture on Tuesday
- Next Lecture on Friday 9 Jan 2009
- Next Exercise session 15 Jan 2009
  - Continuous integration with Cruise Control
- Final exam on 5 Feb 2009
  - Registration via TUMonline
  - Make sure to register in time and not at the last minute.
  - If TUMonline registration is unsuccessful, then use the registration at the Info-Point in the Magistrale.

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

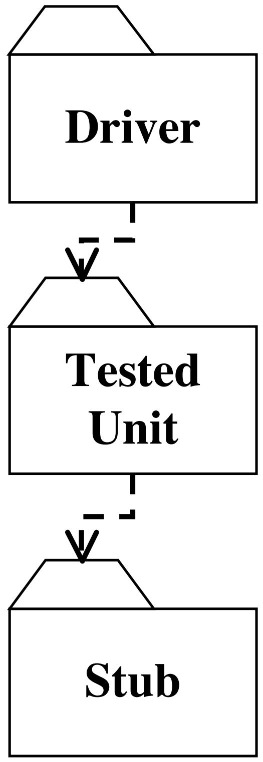
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

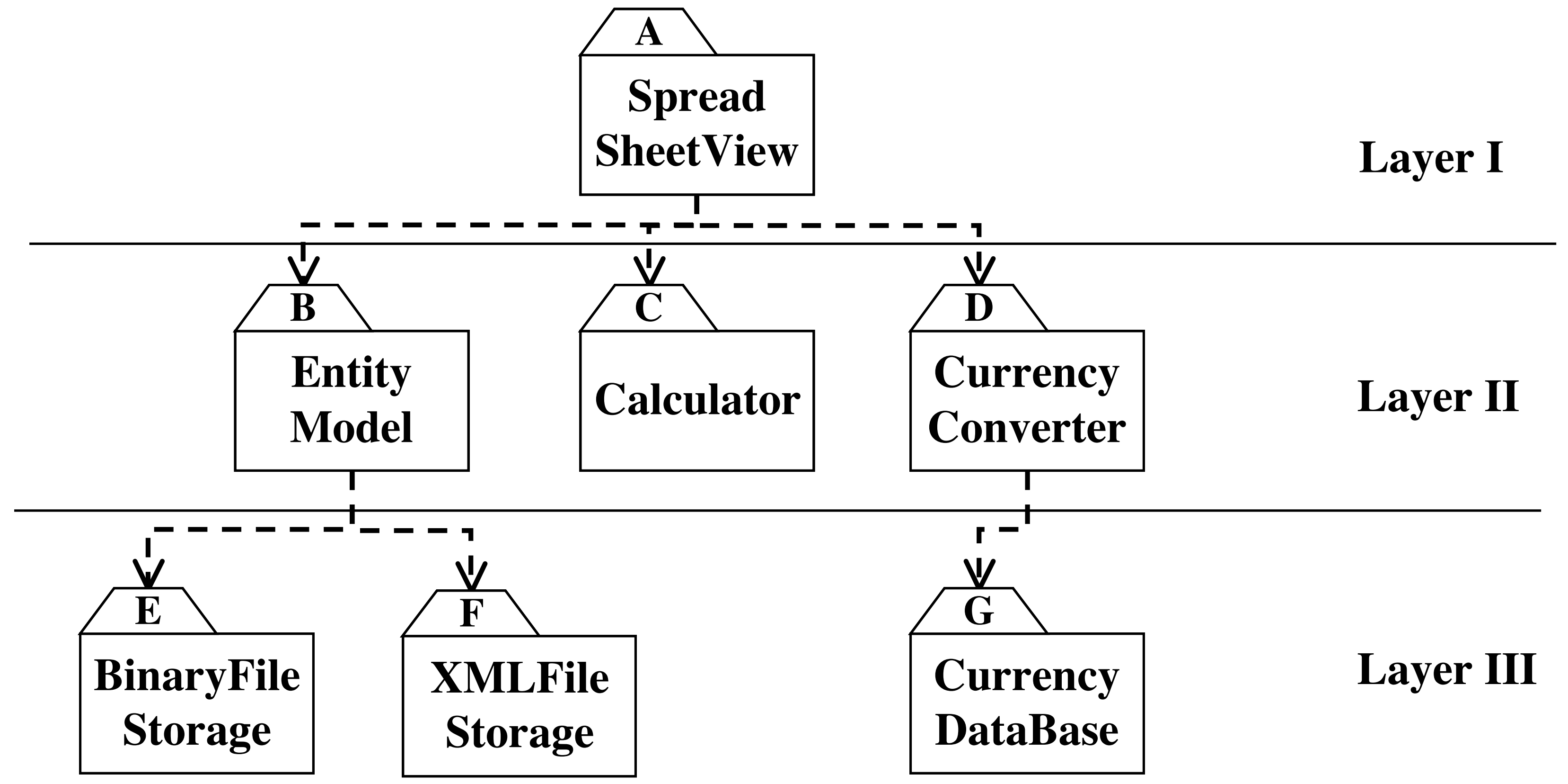
- Unit tests test units and components only in isolation
- Many failures result from faults in the interaction of subsystems
- Off-the-shelf components cannot often not be fully unit tested
- Without integration testing the system test can be very time consuming and costly
    - Failures discovered after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

Definition: Stubs and drivers

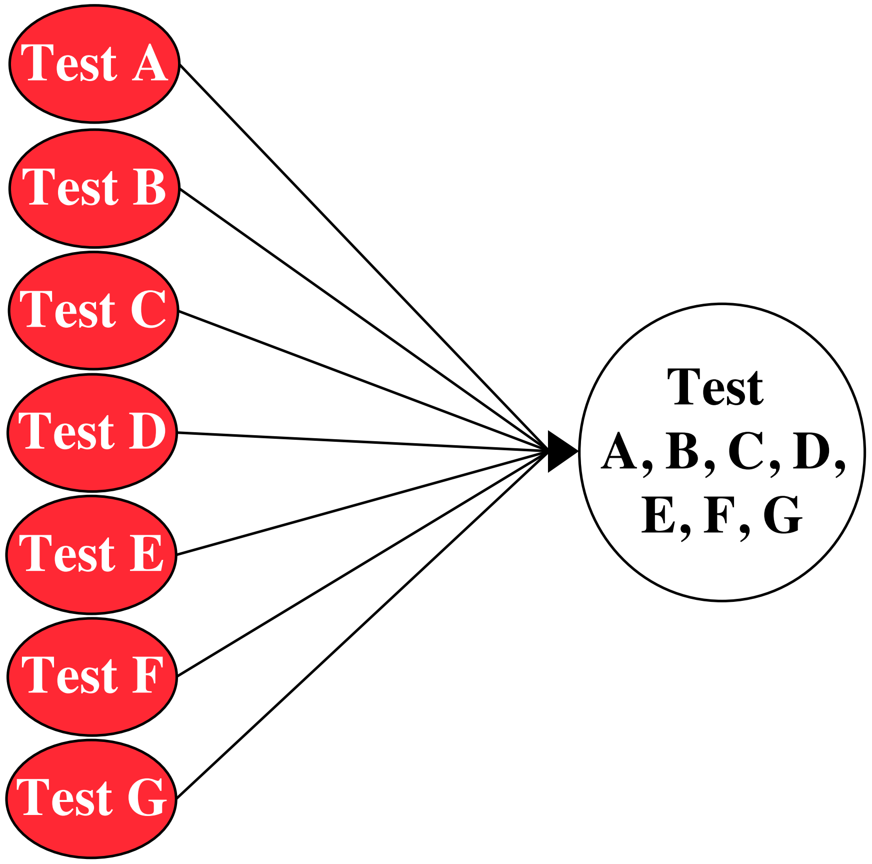
- Stub:
  - A component, the `TestedUnit` depends on which is not yet implemented
  - Usually returns the same value on each call
- Driver:
  - A component which calls the `TestedUnit`
  - Often a replacement for a not yet implemented user interface.

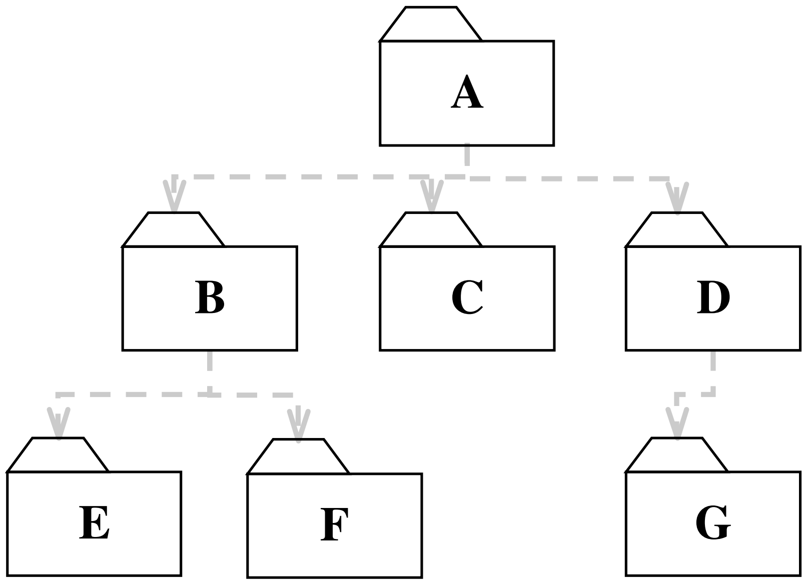


Example: A 3-Layer-Design (Spreadsheet)



Big-Bang Approach





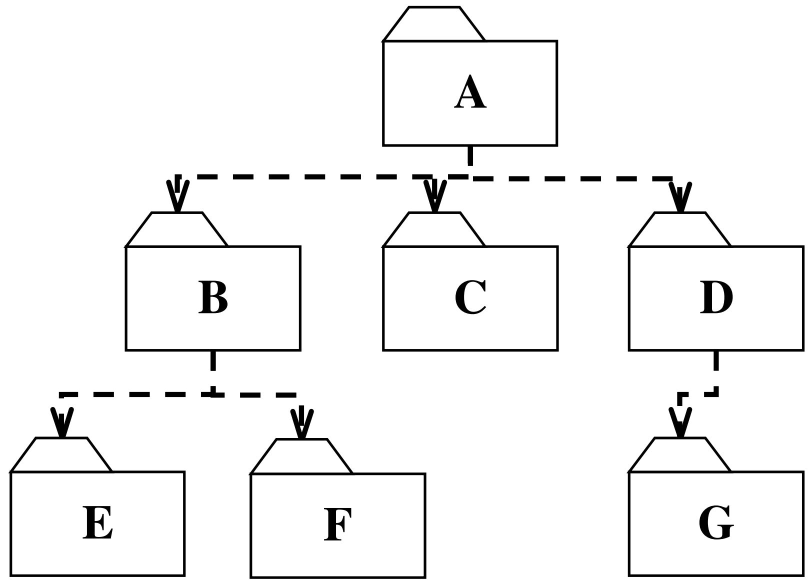
Very bad!
To be avoided at all cost

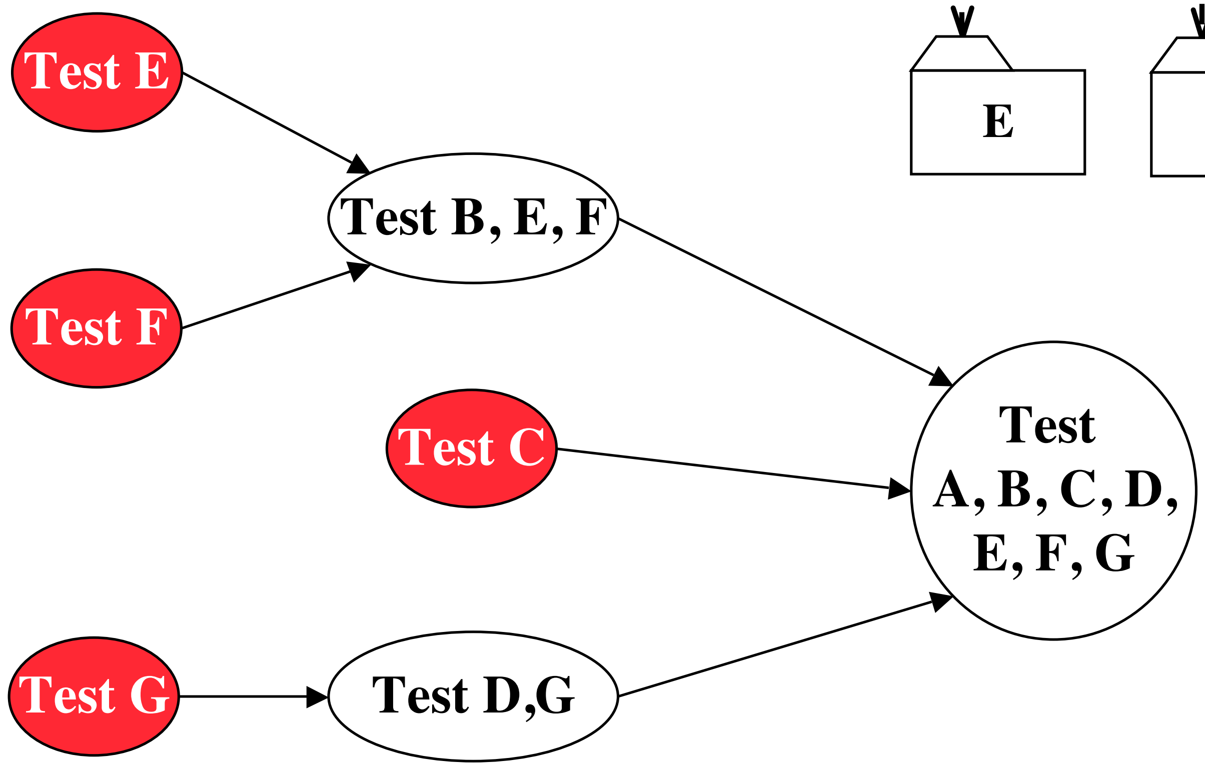
Bottom-up Testing Strategy

1. The subsystems in the lowest layer of the call hierarchy are tested individually
2. Collect the subsystems that call the previously tested subsystems
3. Test the aggregated collection of these subsystems
4. Repeat steps 2 and 3 until all subsystems are included in the test

- Drivers are needed to do bottom-up testing.

Bottom-up Integration





Pros and Cons of Bottom-Up Integration Testing

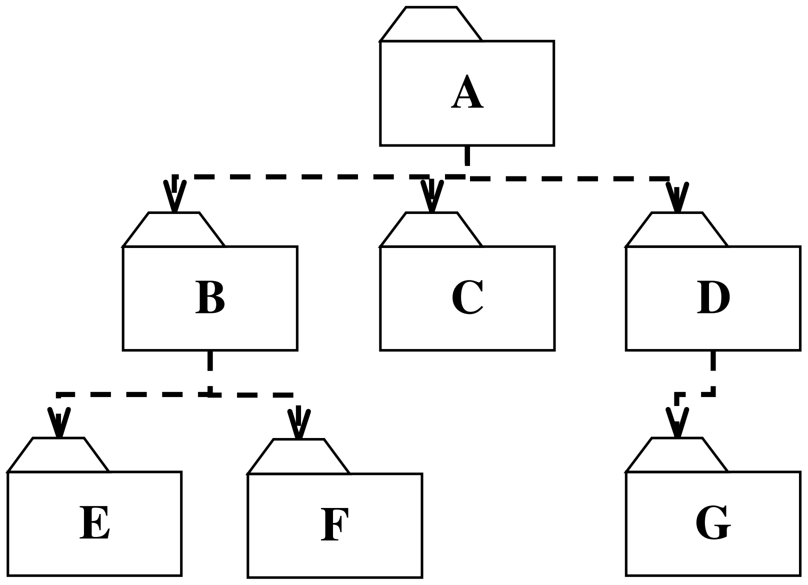
- Con:
  - Tests the most important subsystem (user interface) last
  - Drivers are needed
- Pro
  - No stubs are needed
  - Useful for integration testing of the following systems
    - Systems with strict performance requirements
    - Real-time systems
    - Games.

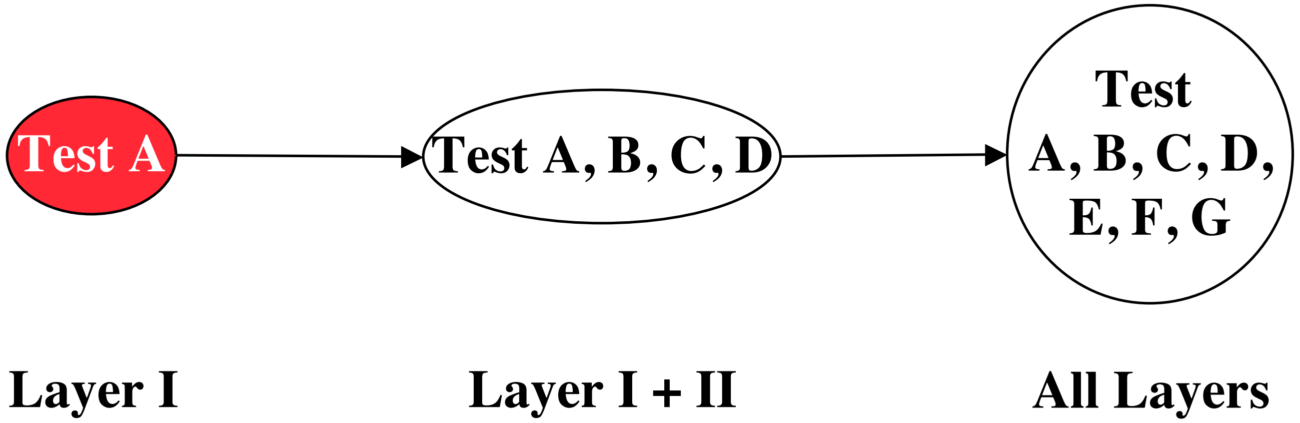
Top-down Testing Strategy

1. Test the top layer or the controlling subsystem first
2. Then combine all the subsystems that are called by the tested subsystems
3. Test this collection of subsystems
4. Repeat steps 2 and 3 until all subsystems are incorporated into the test.

- Stubs are needed to do top-down testing.

Top-down Integration





Pros and Cons of Top-down Integration Testing

Pro:

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

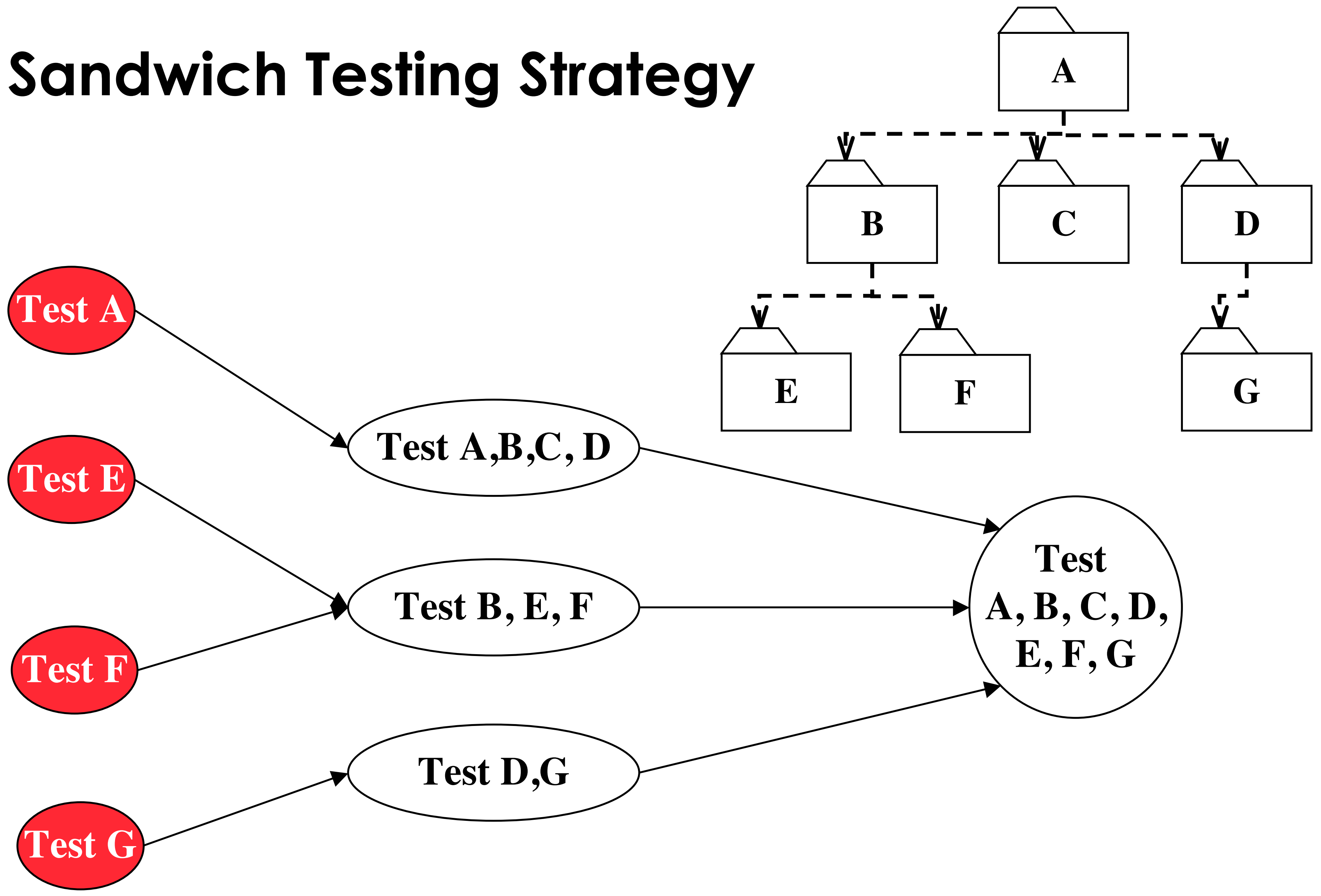
Cons:

- 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.

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



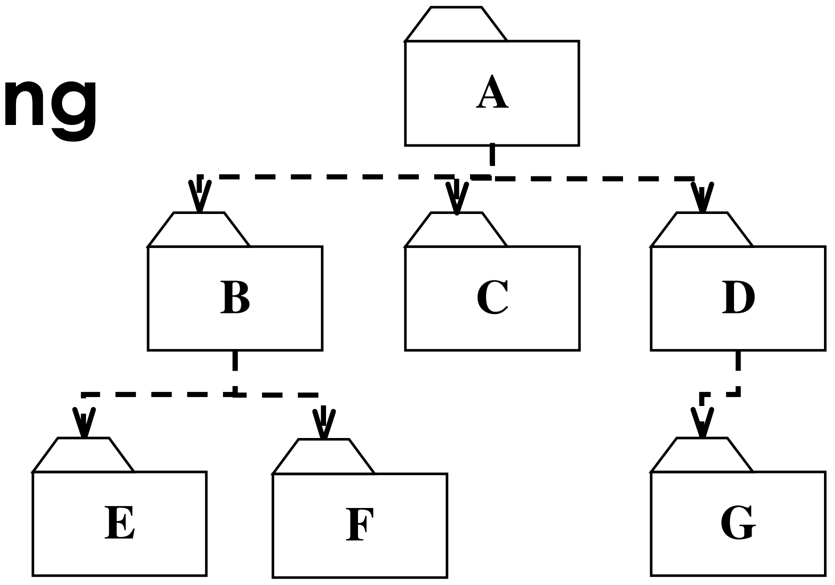
Pros and Cons of Sandwich Testing

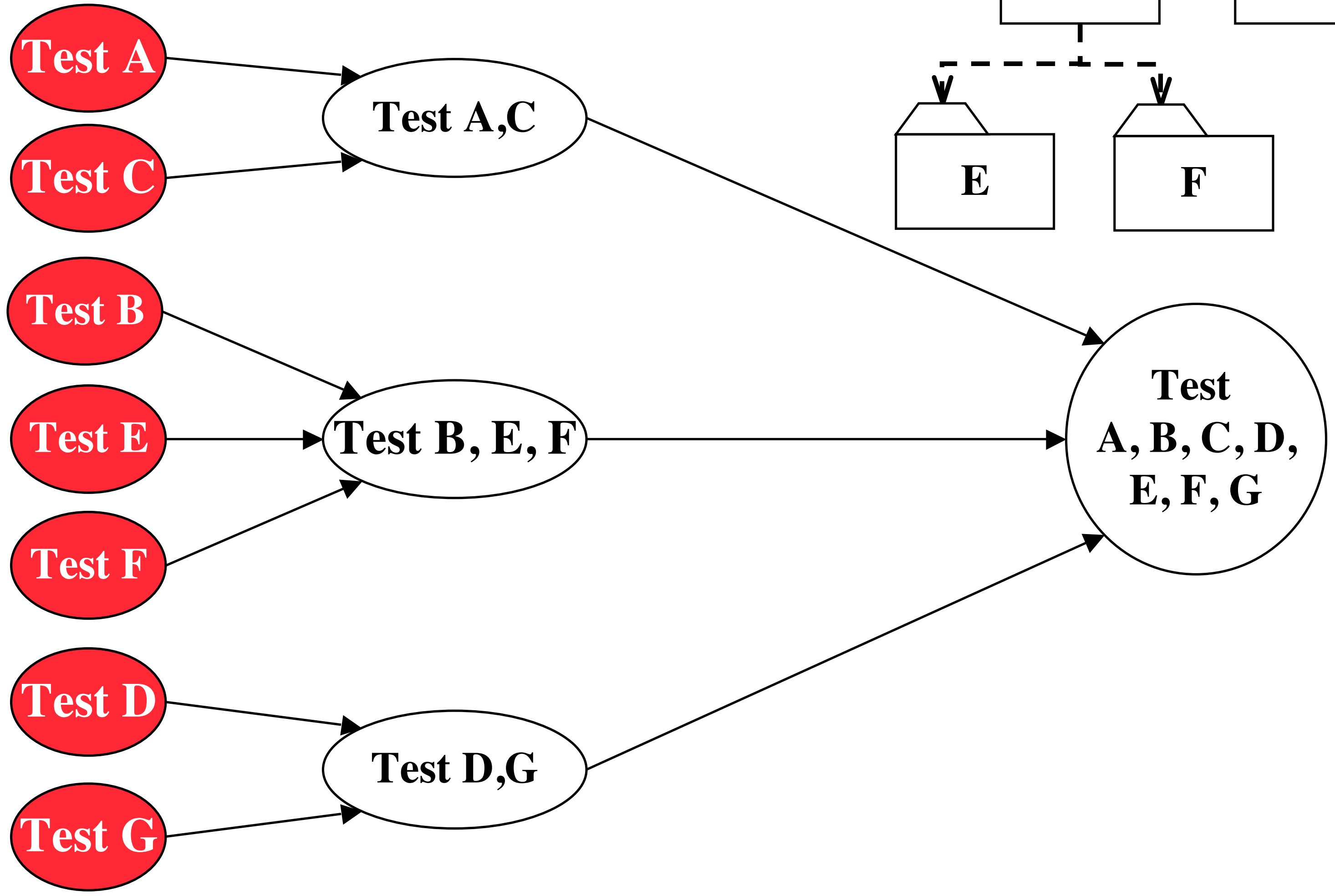
- Top and Bottom Layer Tests can be done in parallel
- Problem: Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy

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



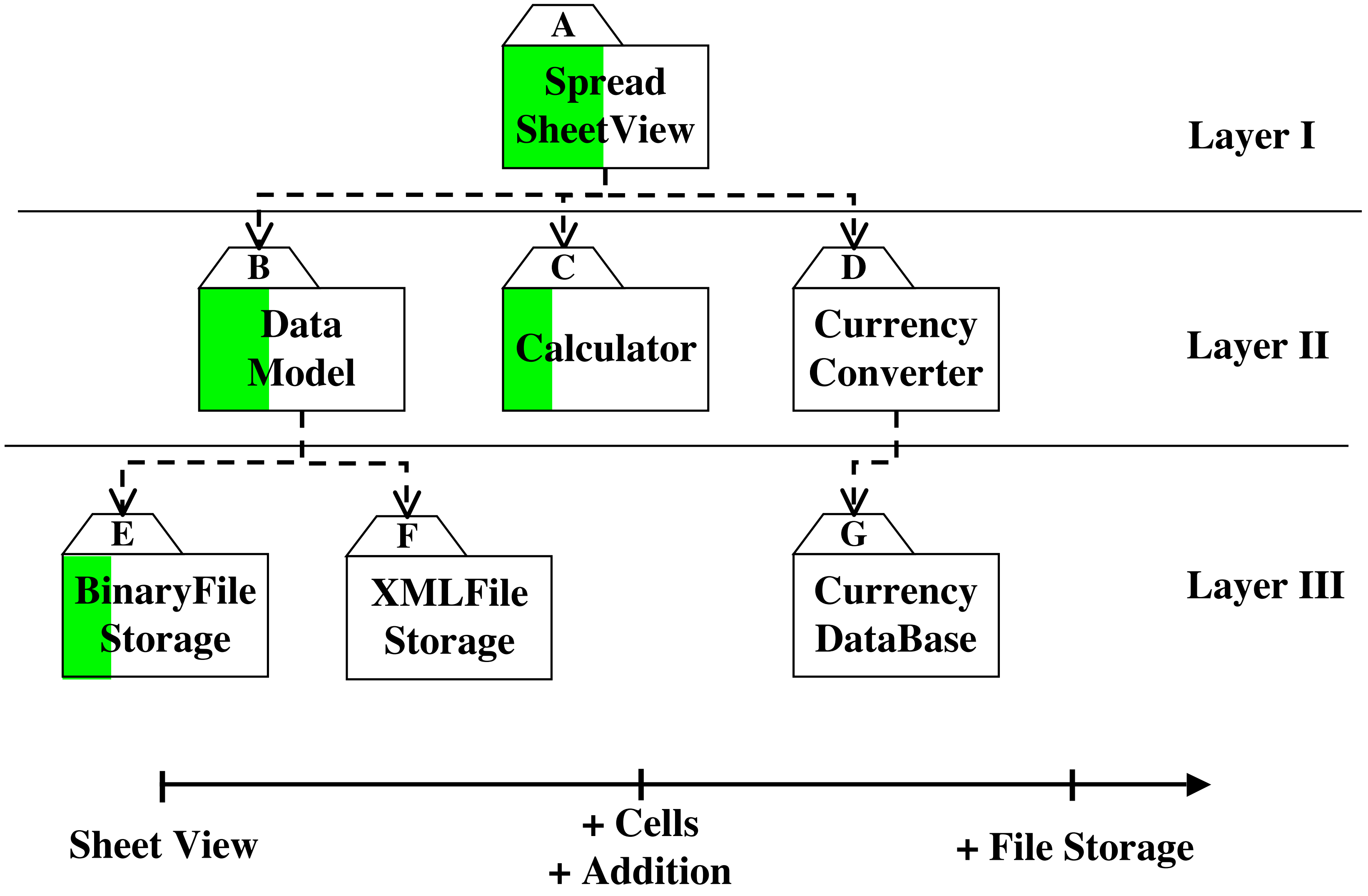


Horizontal vs vertical testing strategies

- Horizontal integration testing:
  - Assumes a hierarchical design
  - Focuses on testing of layers
  - Adds layers incrementally to the test
  - Bottom-up, Top-down and sandwich testing are horizontal testing methods

- Vertical integration testing:
  - Based on scenarios or user stories
  - Takes all the components from the system model that are needed to realize a specific scenario.

Vertical Testing Strategy



Continuous Integration

Continuous integration: A set of software engineering practices to decrease the time needed for integration testing. Consists of these steps:

- Define a build file (make file) as soon as the top level design is defined
- Define and maintain a source code repository
- Check-in and check-out of components
- Perform a unit test after each change of a component
- Do regression testing of the unchanged components
- Automate the Build process:
  - Build a new version of the system after successful testing
  - Make the result of the build process visible for all the developers.

Advantages of Continous Integration

- Immediate unit testing of all changes
- Constant availability of a system for a demo and or a release
- Integration problems are detected and fixed continuously during the duration of the project, not at the last minute
  - Early warning of incompatible interfaces
  - Early warning of conflicting changes
- In short, it avoids all the problems of big bang integration.

Continuous Integration

- Continuous integration is a vertical testing strategy:
  - There is always a runnable and demonstrable version of the system
  - Based on regular builds, first used by Microsoft for the Microsoft Office suite
  - Idea extended by Martin Fowler from Thoughtworks
- Continous integration includes:
  - Software configuration management
  - E-mail notification
  - Automated regression testing tools
  - Continuous build server
  - Views of previous and current builds.

Continous Integration Tools

- Open Source Versions:
  - Cruise Control: http://cruisecontrol.sourceforge.net
    - Java based framwork for a continous build process
  - CruiseControl.NET (Often called CCNET)
    - .NET-based automated continuous integration server
  - Apache Gump: http://gump.apache.org
    - Continuous integration tool for Apache
- Commercial Versions:
  - Cascade: http://www.conifersystems.com/cascade/
    - Provides a "checkpointing" facility by which changes can be built and tested before they are committed
- For more tools, see:
  - http://en.wikipedia.org/wiki/Continuous_Integration

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 environ-ment

Testing has many activities

Establish the test objectives

Design the test cases

Write the test cases

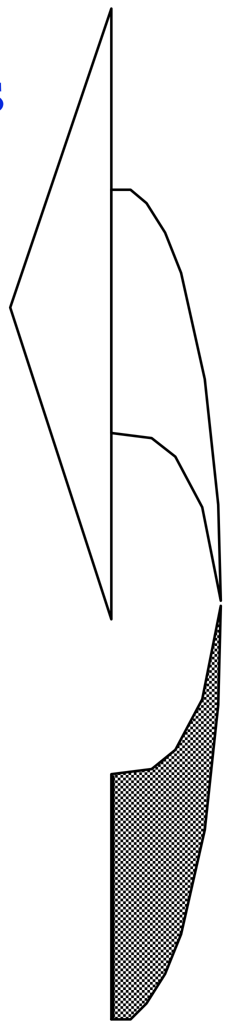
Test the test cases

Execute the tests

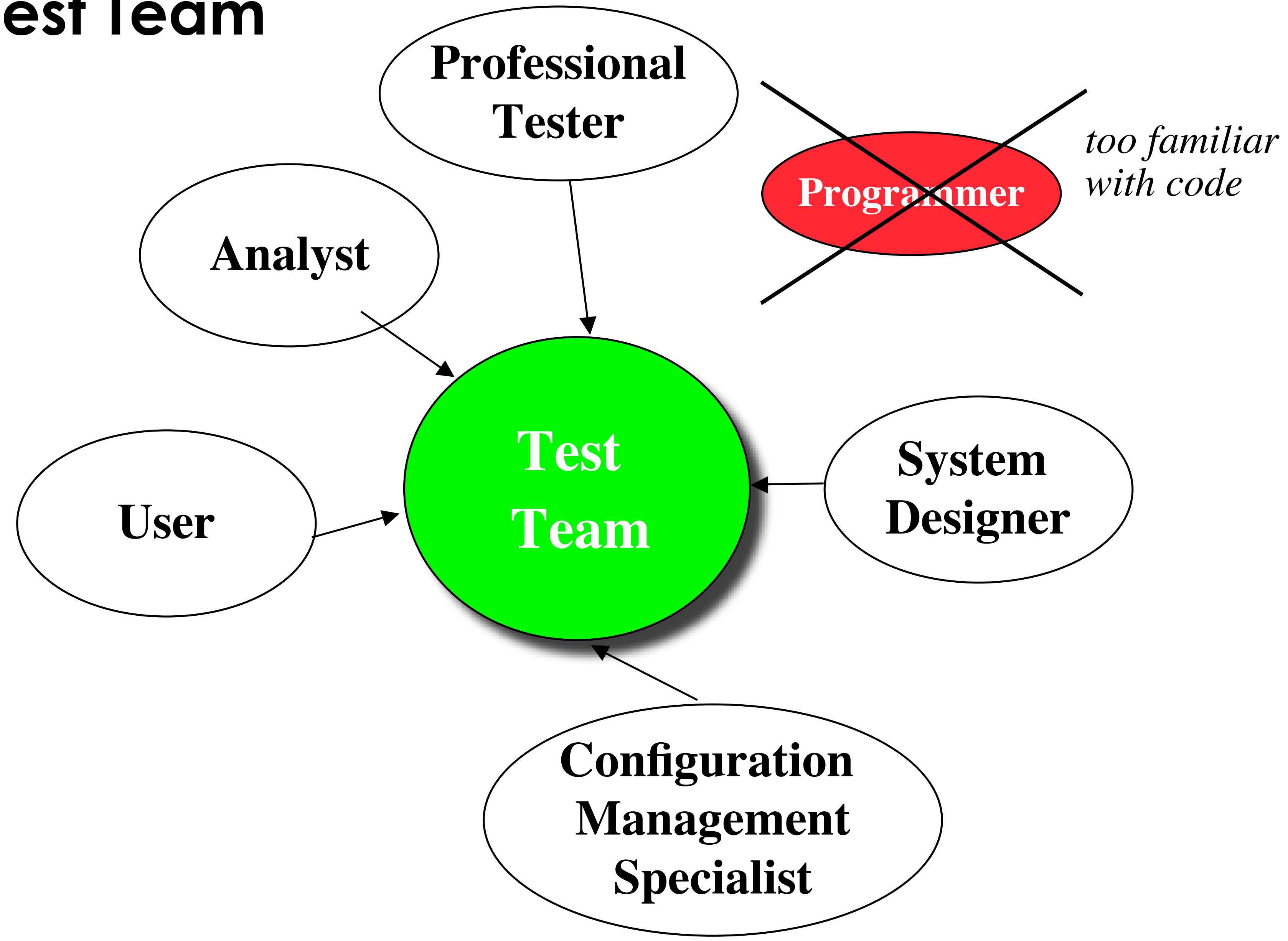
Evaluate the test results

Change the system

Do regression testing



Test Team


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

The 4 Testing Steps

1. Select <u>what</u> has to be tested
   - Analysis: Completeness of requirements
   - Design: Cohesion
   - Implementation: Source code
2. Decide <u>how</u> the testing is done
   - Review or code inspection
   - Proofs (Design by Contract)
   - Black-box, white box,
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)
3. Develop <u>test cases</u>
   - 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 <u>test oracle</u>
   - 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
- Design patterns can be used for integration testing
- Testing has its own lifecycle

Additional Reading

- JUnit Website www.junit.org/index.htm
- J. Thomas, M. Young, K. Brown, A. Glover, Java Testing Patterns, Wiley, 2004
- Martin Fowler, Continous Integration, 2006
    - http://martinfowler.com/articles/continuousIntegration.html
    - Original version of the article:
    - http://martinfowler.com/articles/originalContinuousIntegration.html
- D. Saff and M. D. Ernst, An experimental evaluation of continuous testing during development *Int. Symposium on Software Testing and Analysis*, Boston July 12-14, 2004, pp. 76-85
    - A controlled experiment shows that developers using continuous testing were three times more likely to complete the task before the deadline than those without.

Merry Christmas and
a Happy New Year!