System Testing

Software Engineering 1
Lecture 16

Bernd Bruegge, Ph.D.
Applied Software Engineering
Technische Universitaet Muenchen

Remaining Lecture and Exercise Schedule

- Jan 24 (Today), HS 1
  - System Testing, Software Lifecycle
- Jan 30 - Jan 31, HS 1
  - Software Lifecycle II, Methodologies
- Feb 6 - Feb 7, HS 1
  - NEW: How to Present and Review a Software System
    - Heuristics and Suggestions
  - Methodologies II
- Feb 8, 16:30-18:00 Room 00.08.038
  - Miniproject presentations, Part 1
- Feb 9, 16:30-18:00 Room 00.08.038
  - Miniproject presentations, Part 2

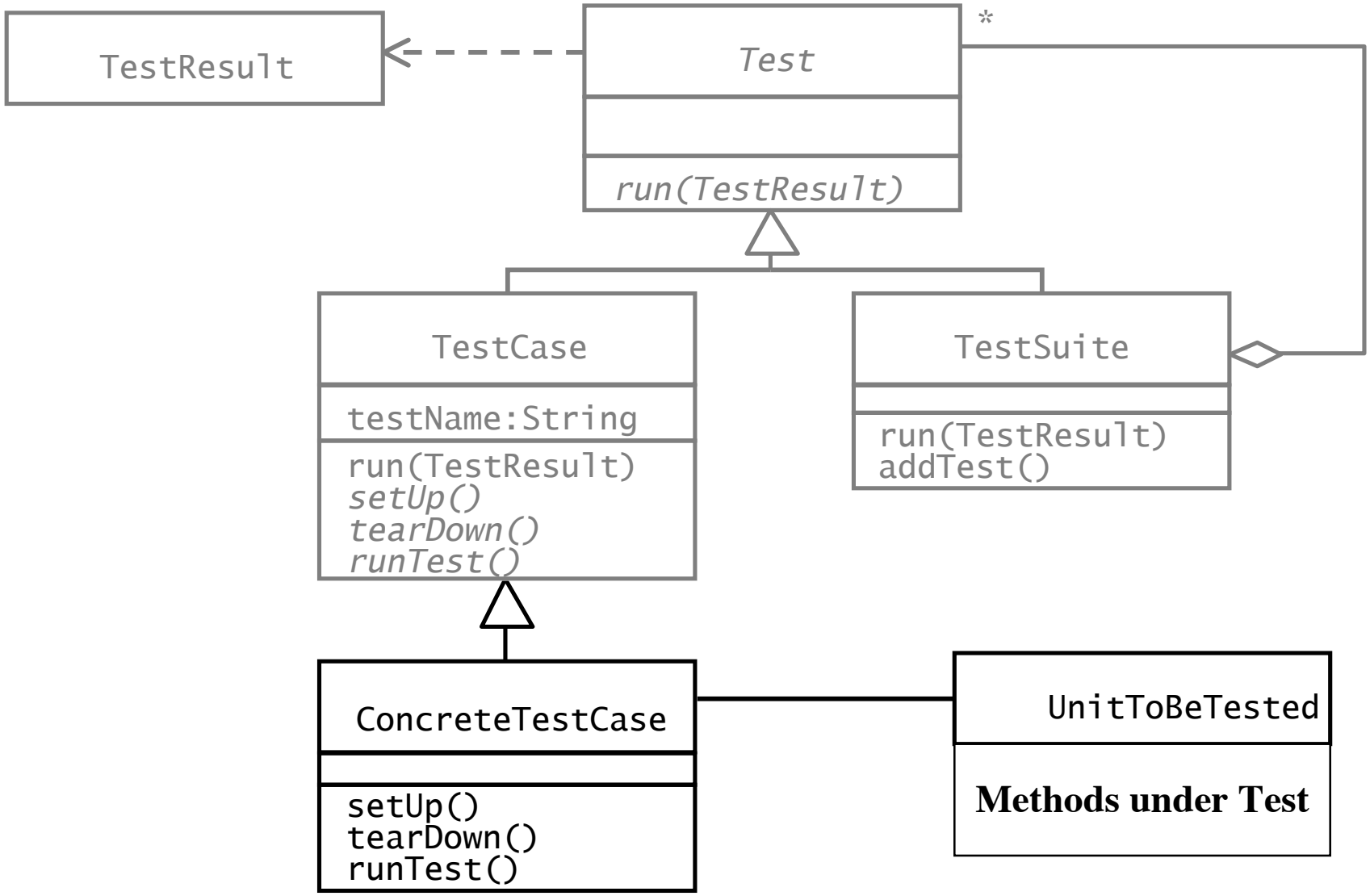
Overview

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

JUnit: Overview

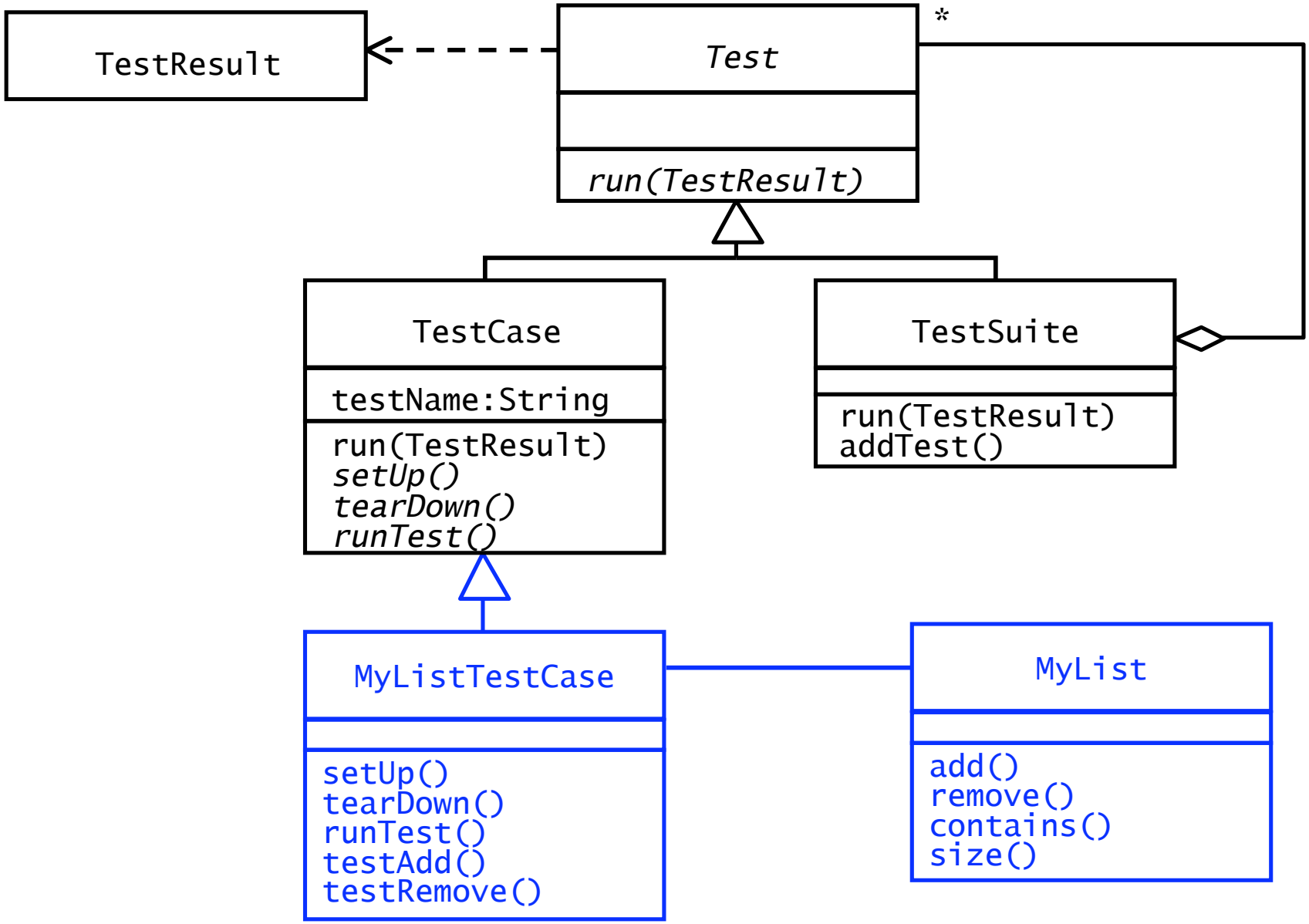
- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Written by Kent Beck and Erich Gamma
- Written with "test first" and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring
- JUnit is Open Source
  - www.junit.org
  - JUnit Version 4, released Mar 2006

JUnit Classes



An example: Testing MyList

- Unit to be tested
  - MyList
- Methods under test
  - add()
  - remove()
  - contains()
  - size()
- Concrete Test case
  - MyListTestCase


TestResult
Test
*
run(TestResult)
TestCase
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
run(TestResult)
addTest()
MyListTestCase
setUp()
tearDown()
runTest()
testAdd()
testRemove()
MyList
add()
remove()
contains()
size()

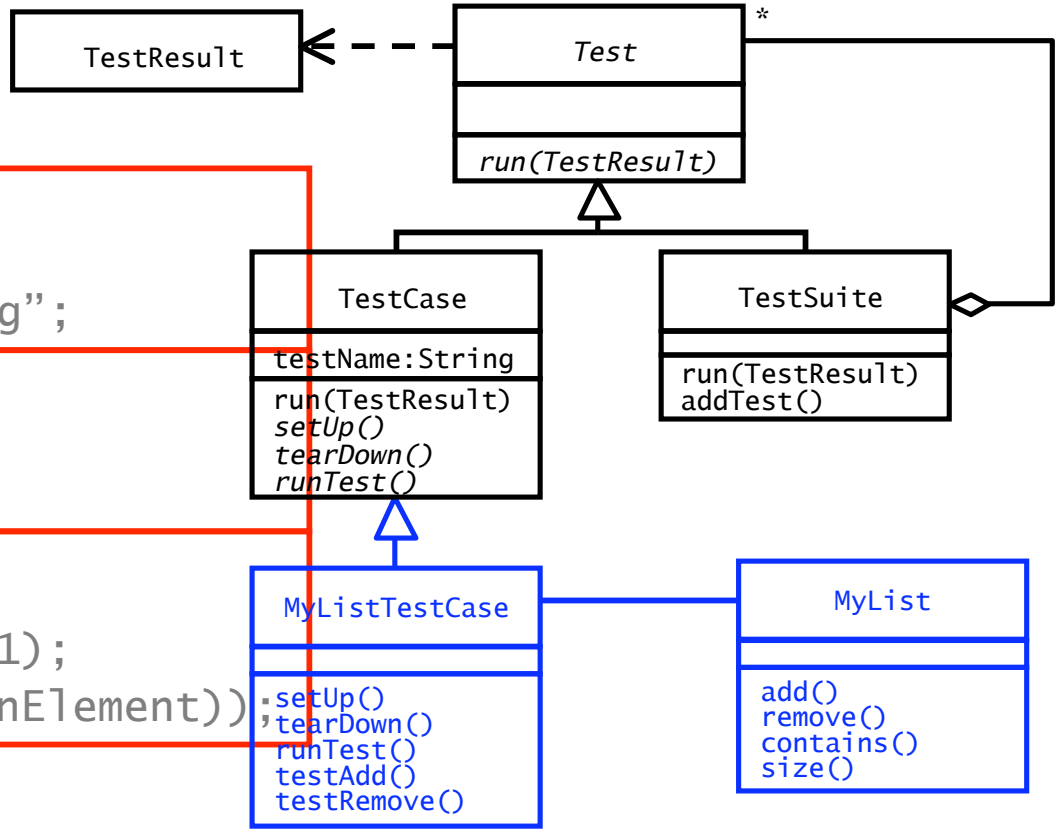
Writing TestCases in JUnit

```
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
   super(name);
}
public void testAdd() {
   // Set up the test
   List aList = new MyList();
   String anElement = "a string";

   // Perform the test
   aList.add(anElement);

   // Check if test succeeded
   assertTrue(aList.size() == 1);
   assertTrue(aList.contains(anElement));
}
protected void runTest() {
   testAdd();
}
}
```



Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
   aList = new MyList();
   anElement = “a string”;
}

public void testAdd() {
   aList.add(anElement);
   assertTrue(aList.size() == 1);
   assertTrue(aList.contains(anElement));
}

public void testRemove() {
   aList.add(anElement);
   aList.remove(anElement);
   assertTrue(aList.size() == 0);
   assertFalse(aList.contains(anElement));
}
```

Test Fixture

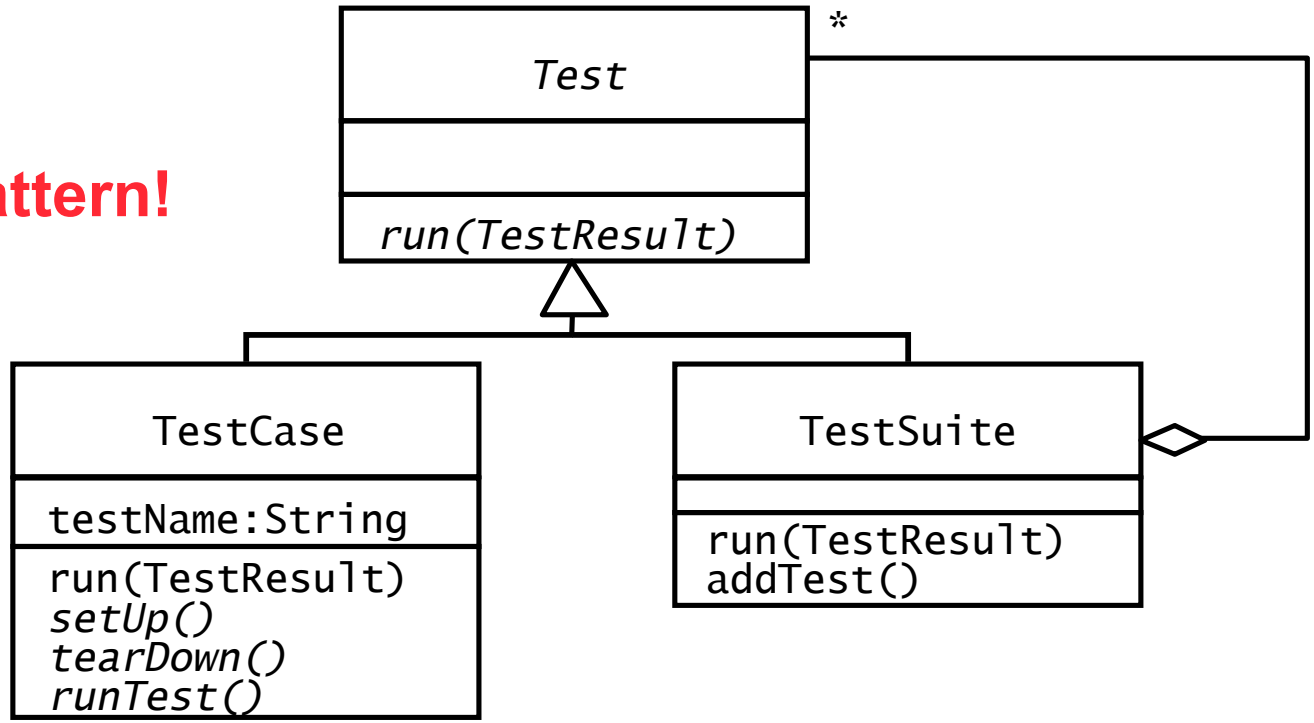
Test Case

Test Case

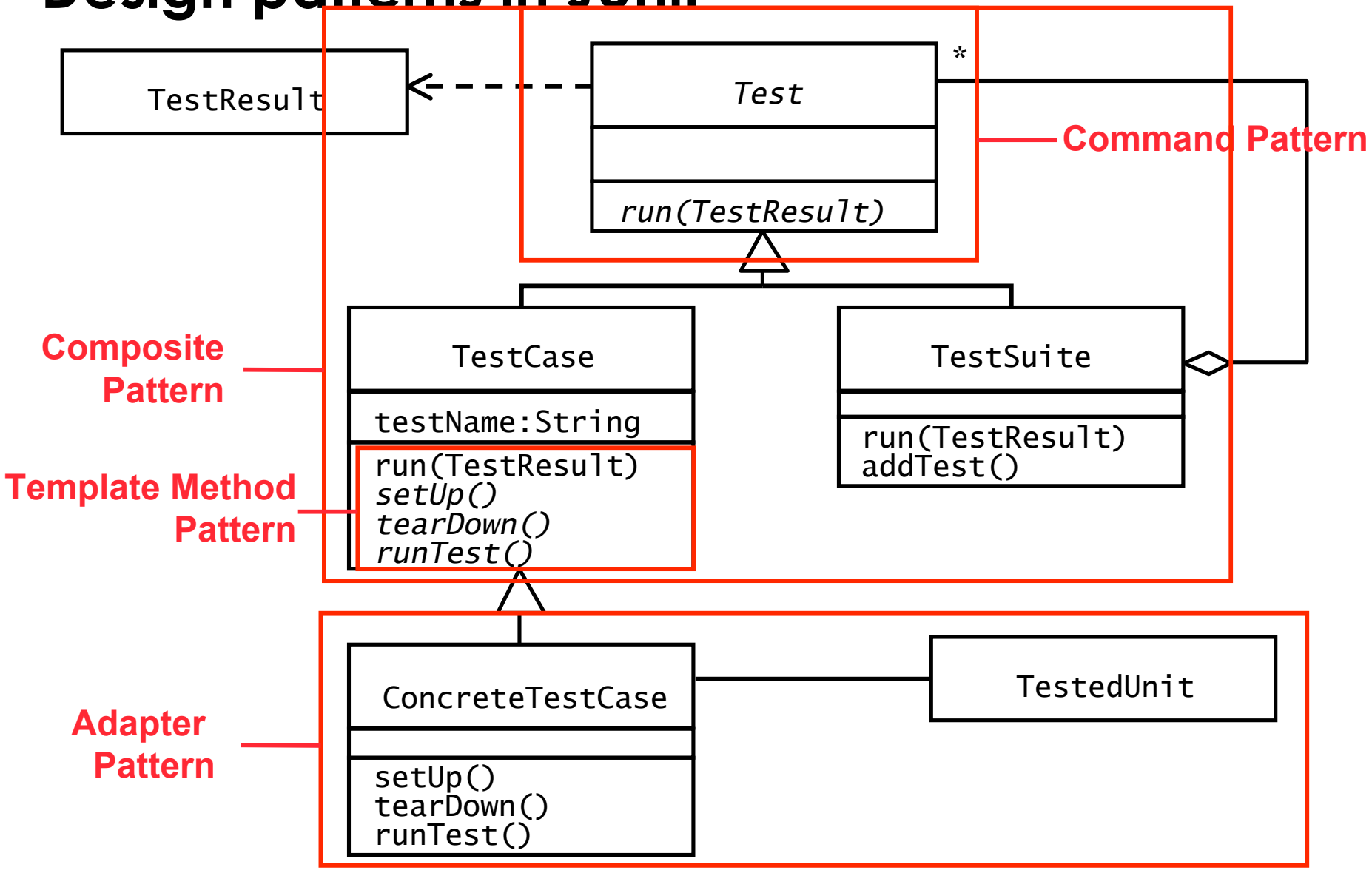
Collecting TestCases into TestSuites

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new MyListTest("testAdd"));
  suite.addTest(new MyListTest("testRemove"));
  return suite;
}
```

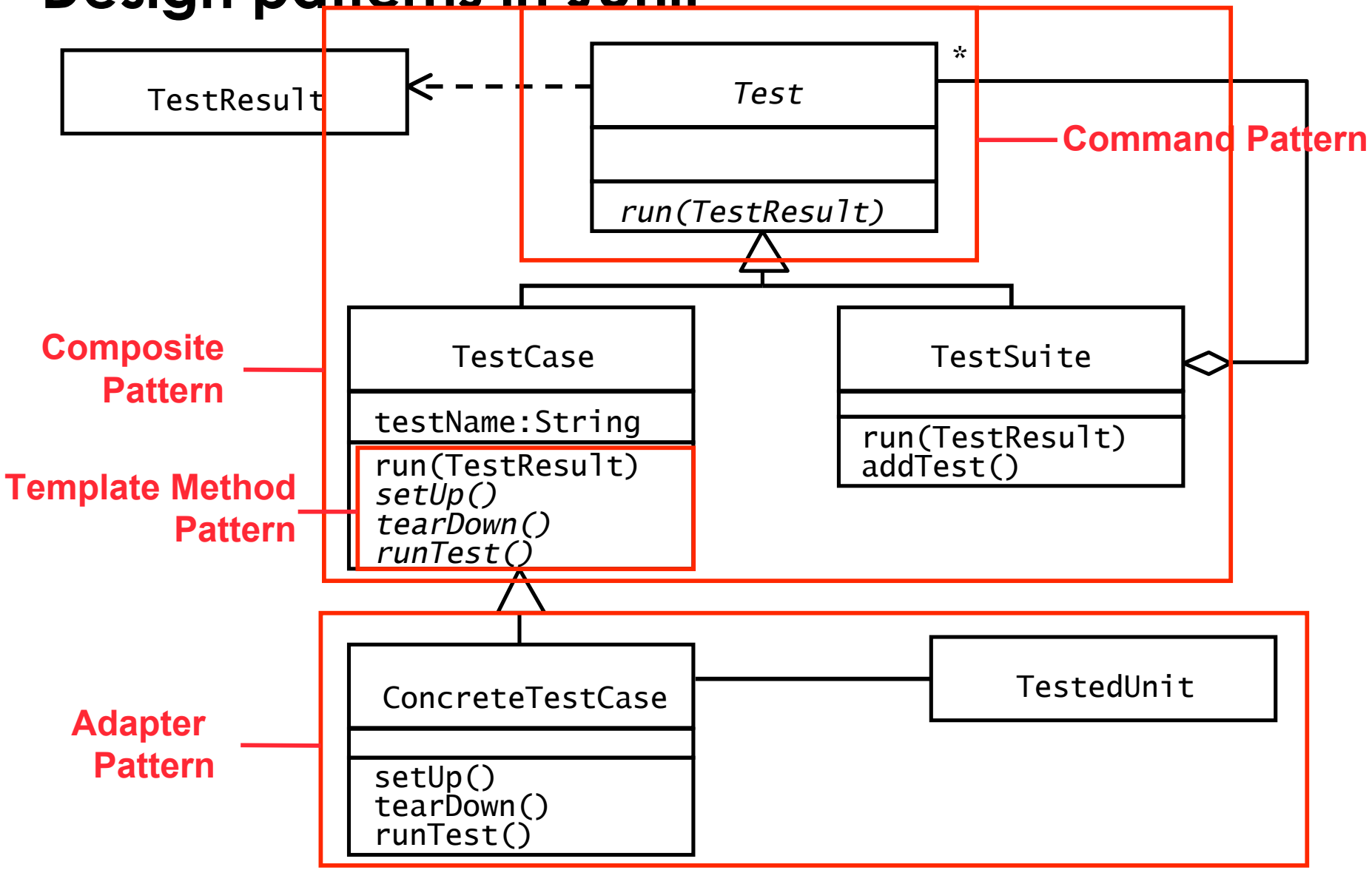
Composite Pattern!



Design patterns in JUnit



Design patterns in JUnit



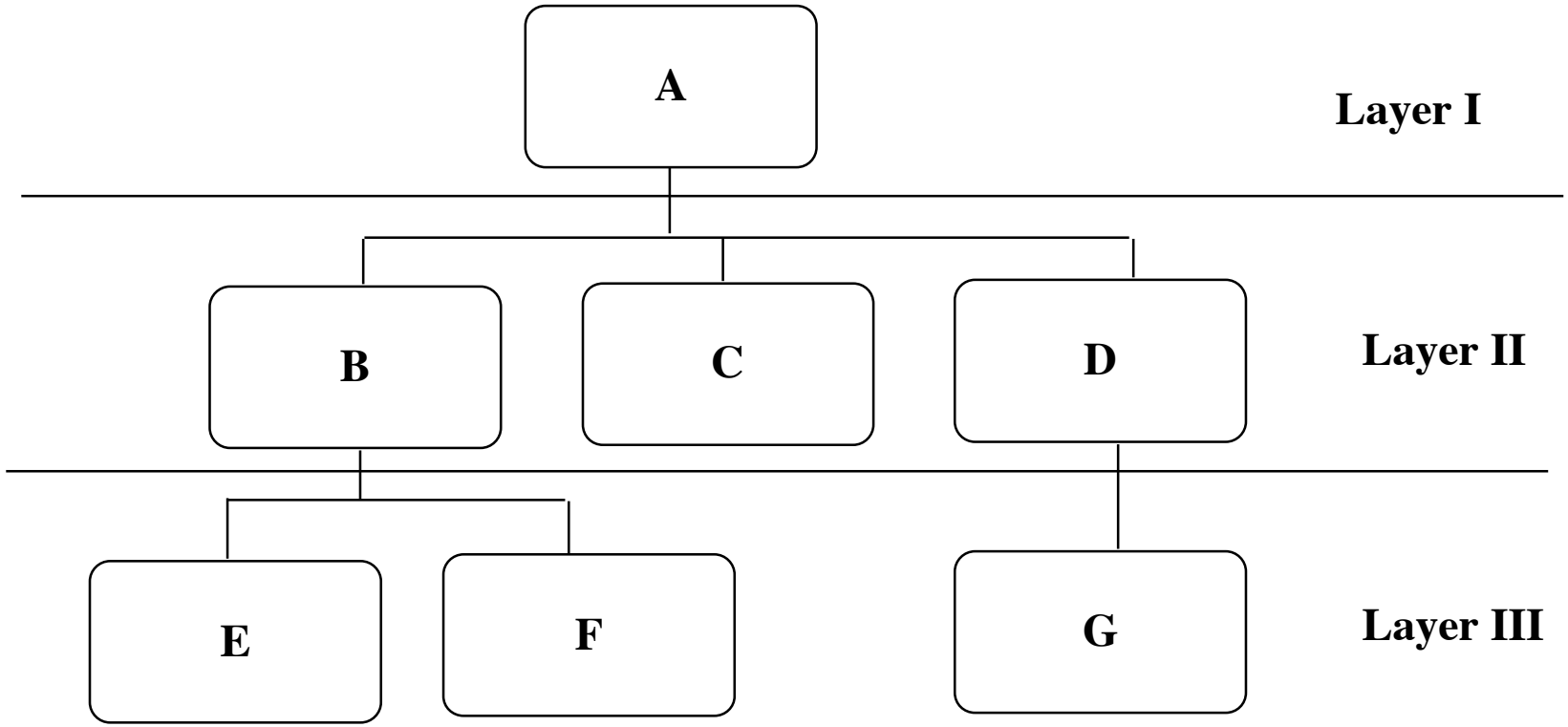
Other JUnit features

- Textual and GUI interface
  - Displays status of tests
  - Displays stack trace when tests fail
- Integrated with Maven and Continuous Integration
  - maven.apache.org
    - Build and Release Management Tool
  - Maven.apache.org/continuum
    - Continous integration server for Java programs
  - All tests are run before release (regression tests)
  - Test results are advertised as a project report
- Many specialized variants
  - Unit testing of web applications
  - J2EE applications

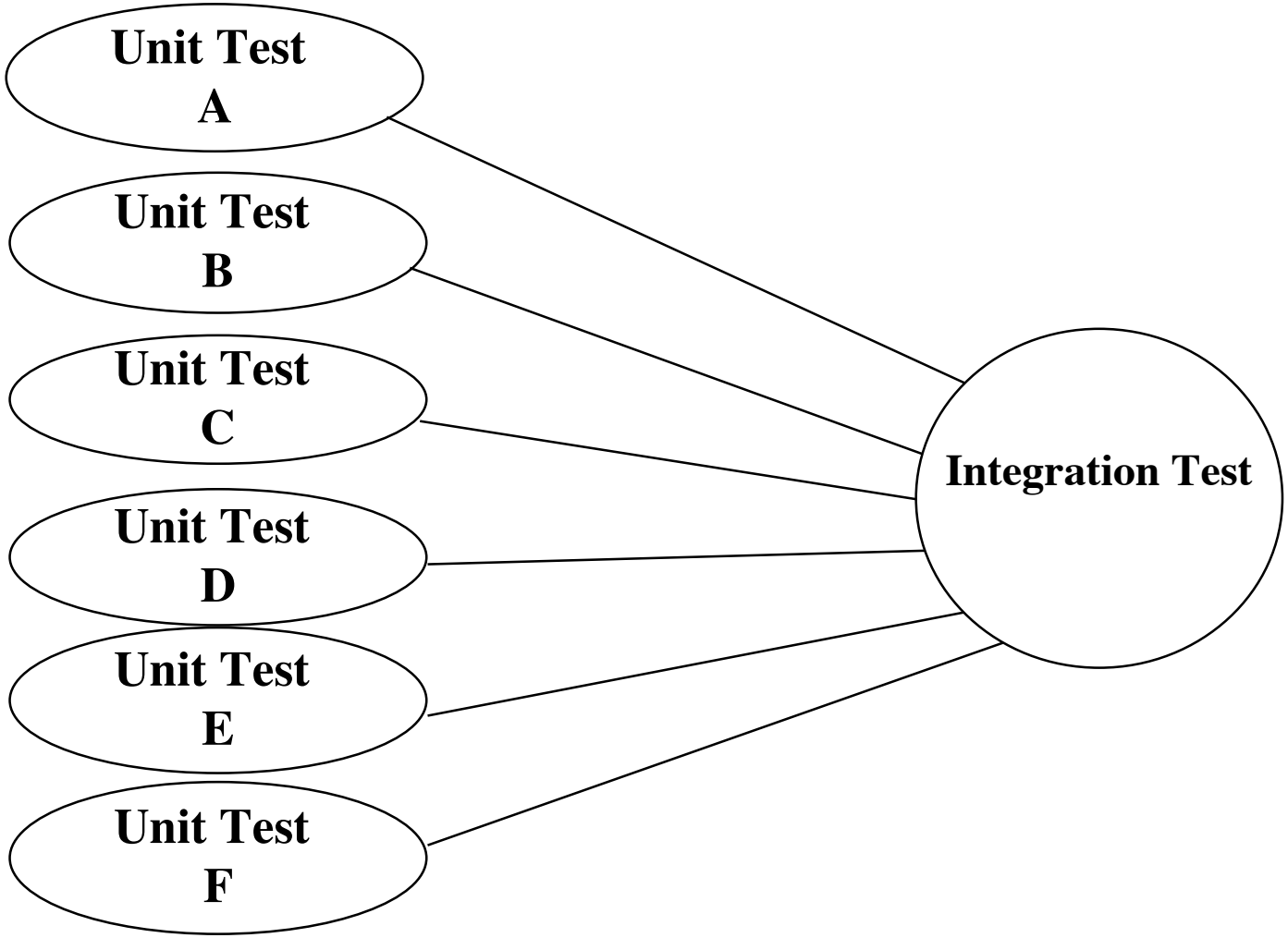
Integration Testing

- 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.

Example: A 3-Layer-Design

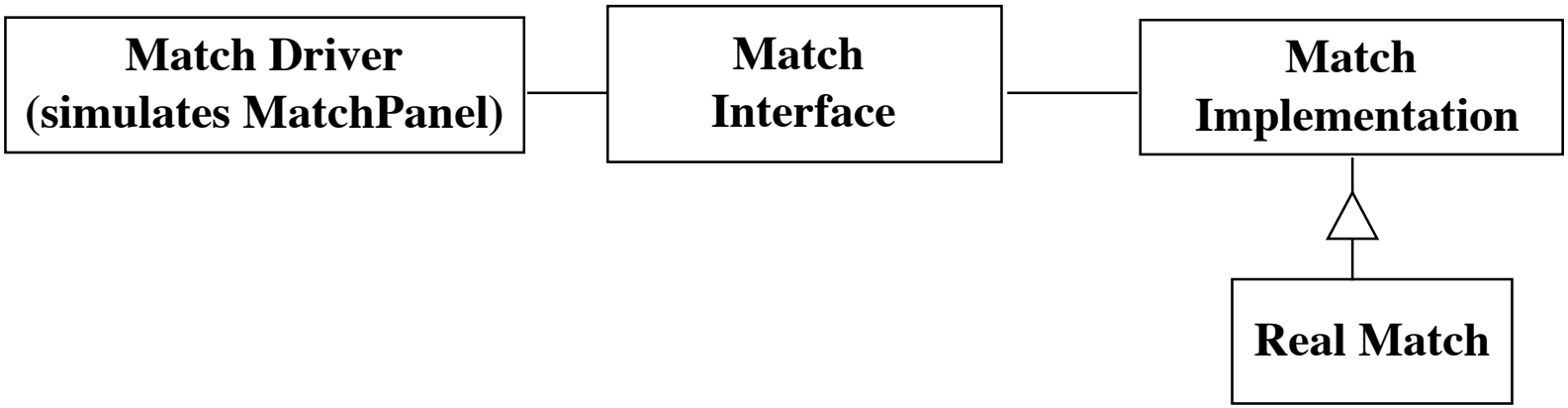


Integration Testing: Big-Bang Approach

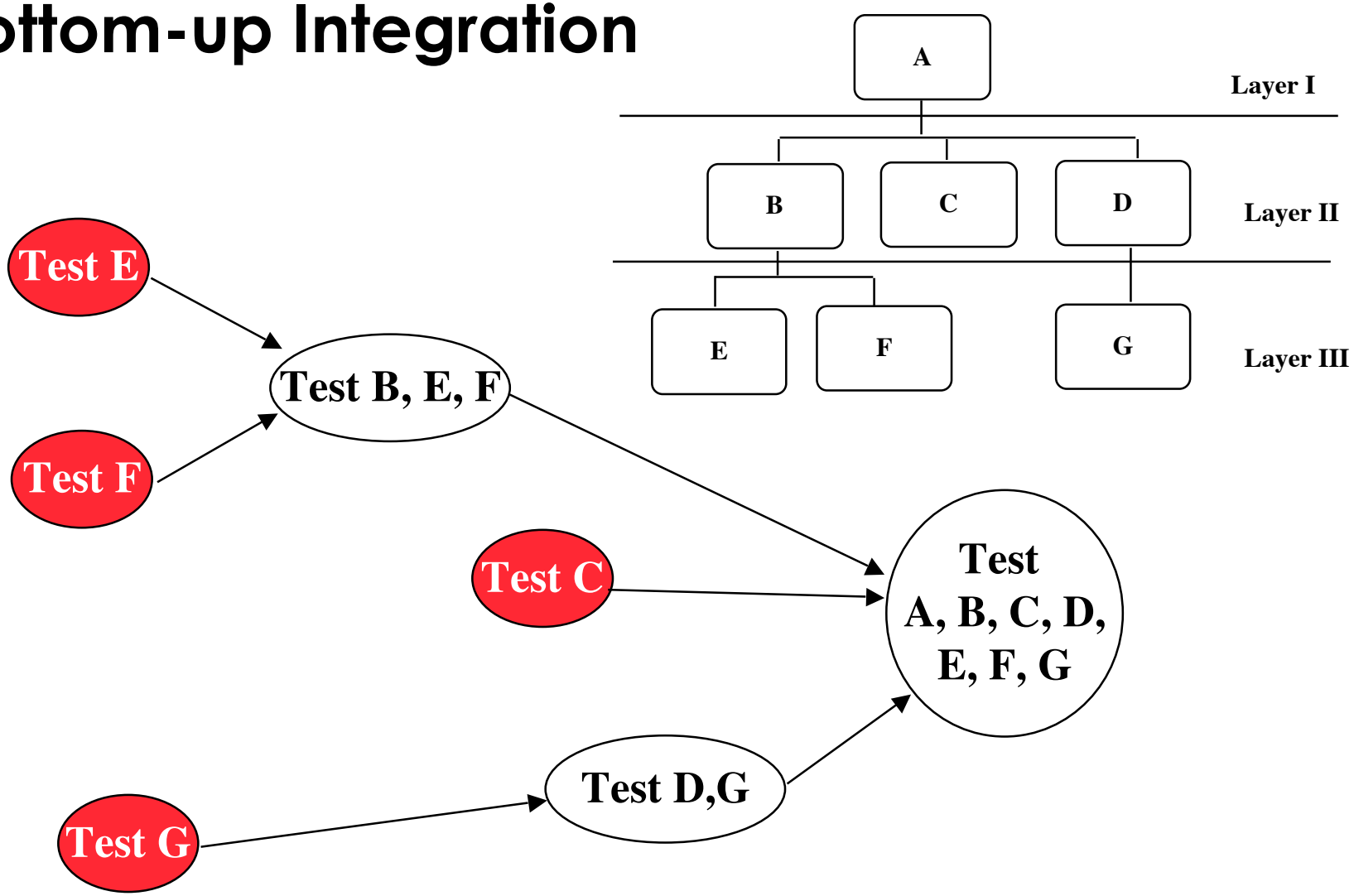


Bottom-up Testing Strategy

- 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 repeated until all subsystems are included
- Special program needed for testing("Test Driver"): A routine that calls a subsystem and passes a test case to it



Bottom-up Integration



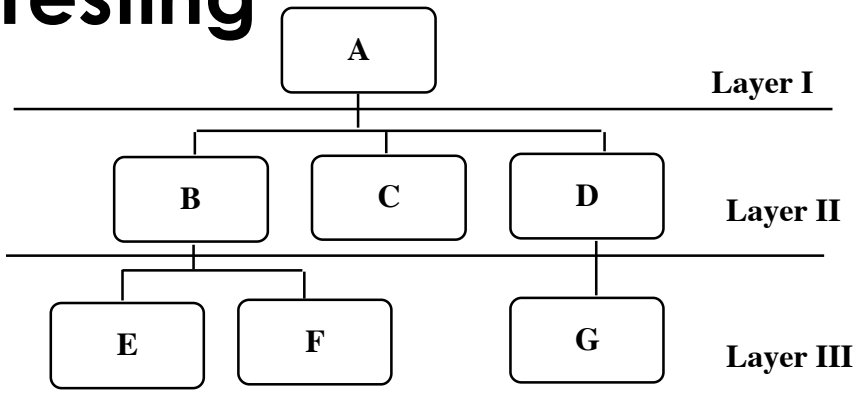
Pros and Cons of Bottom-Up Integration Testing

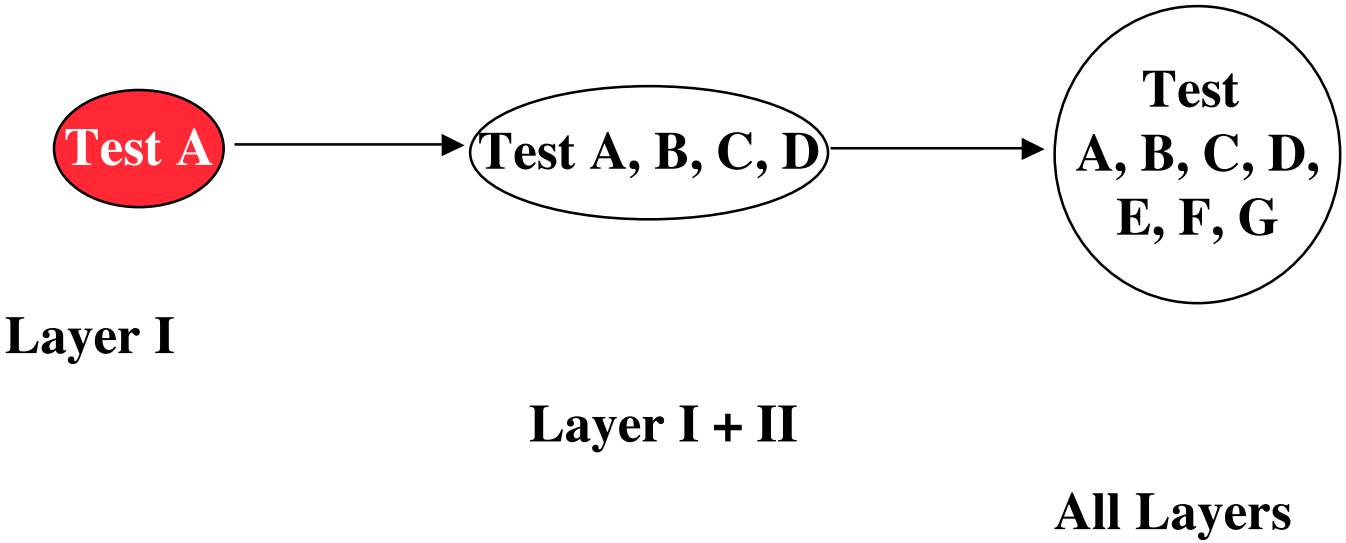
- Con:
  - Tests the most important subsystem (user interface) last
- Pro:
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements.

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





Pros and Cons of Top-down Integration Testing

Pro

- Test cases can be defined in terms of the functionality of the system (functional requirements)

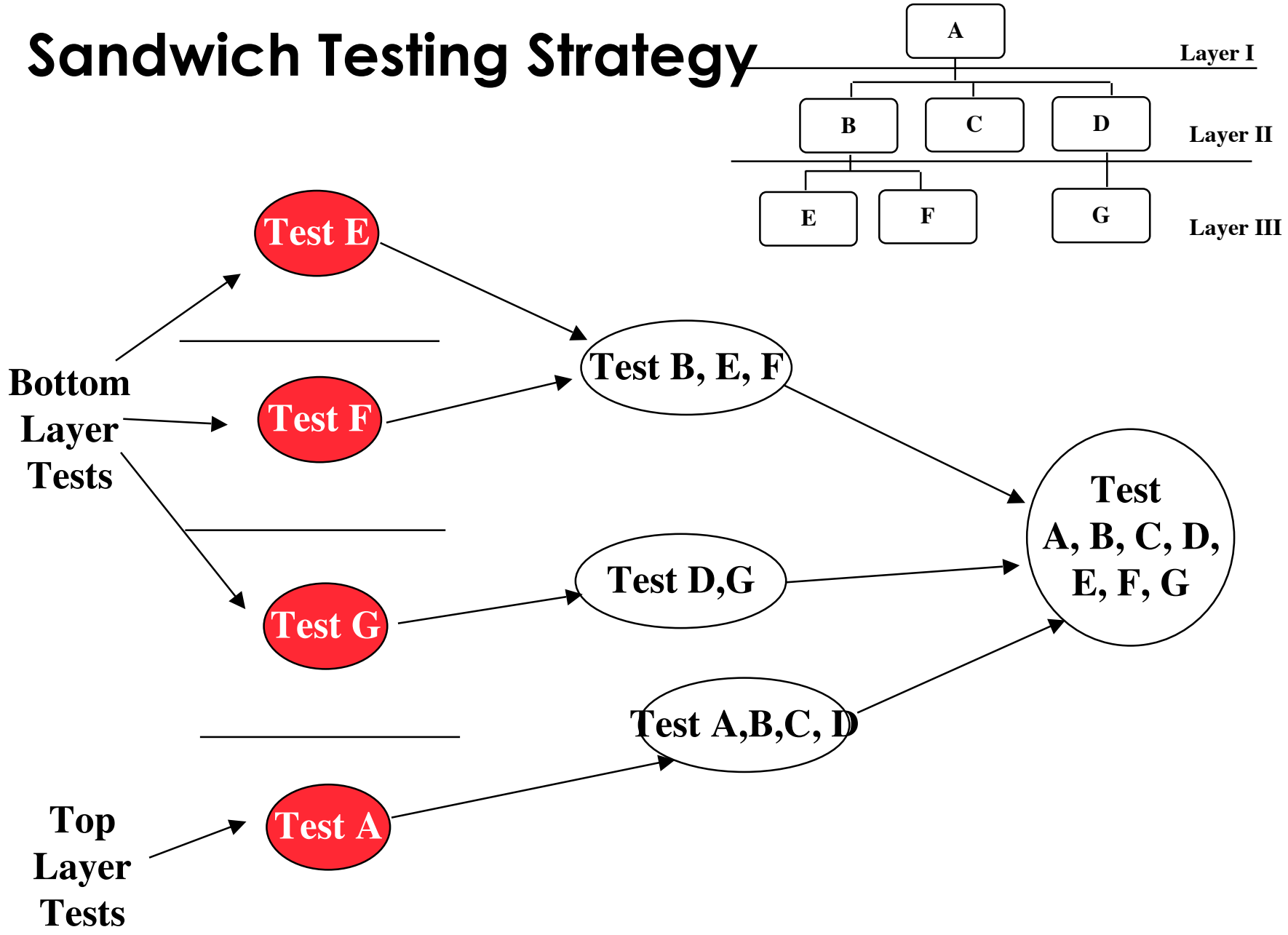
Cons

- 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.

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

Sandwich Testing Strategy



Pros and Cons of Sandwich Testing

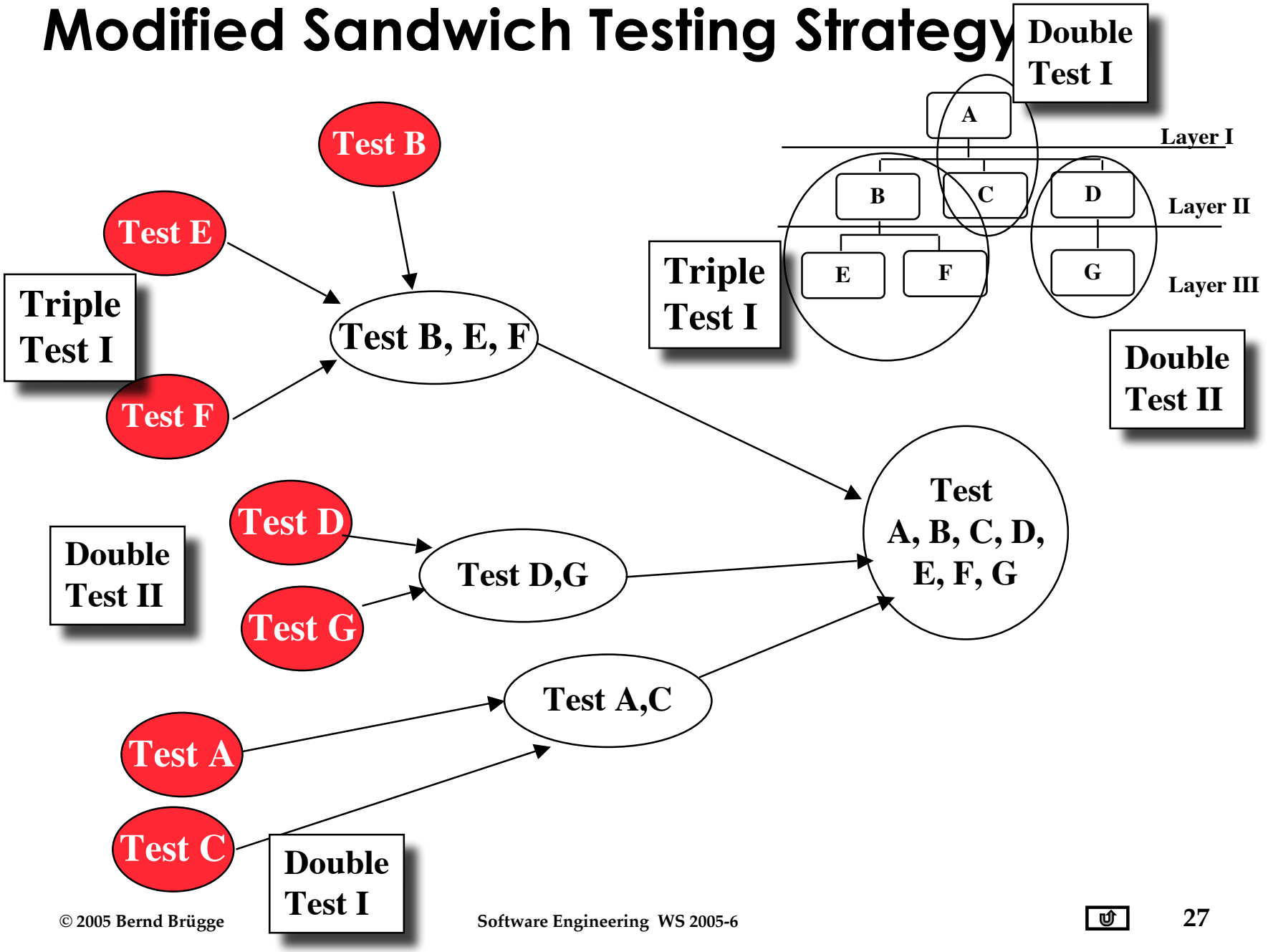
- Top and Bottom Layer Tests can be done in parallel

- Problem: Does not test the individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy

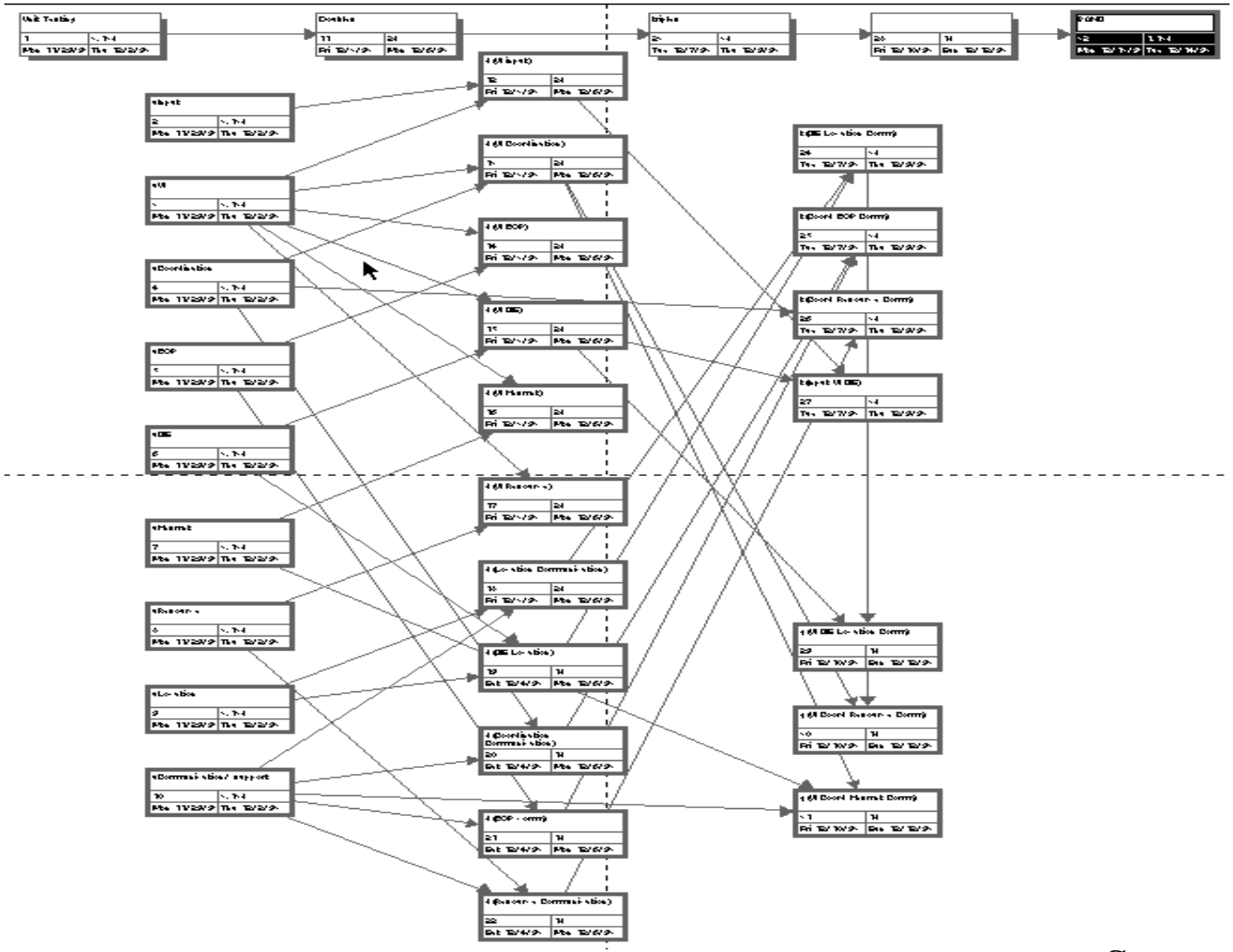
Modified Sandwich Testing Strategy

- 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 1 23 2007



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.

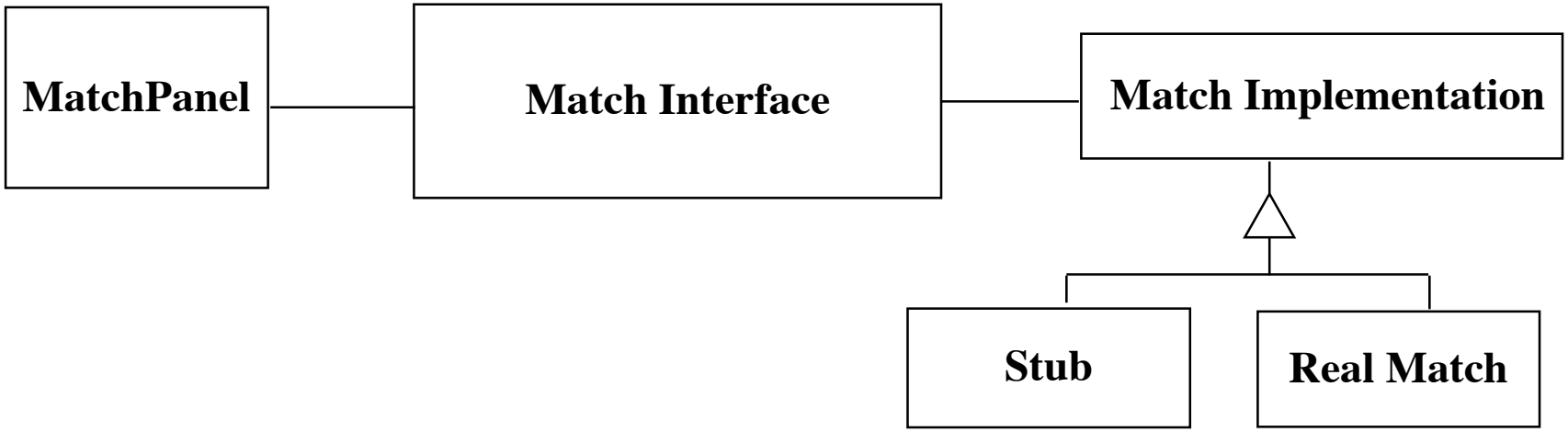
The primary *goal of integration testing is to identify errors* in the (current) component *configuration*.

Which Integration Strategy should you use?

- Factors to consider
  - Amount of test overhead (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

Using the Bridge design 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



System Testing 1 17 2006

- Impact of requirements on system testing:
  - 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
- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

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.

Structure 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.

Performance Testing

Goal: Try to break the subsystems

- 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:
  - 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

Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

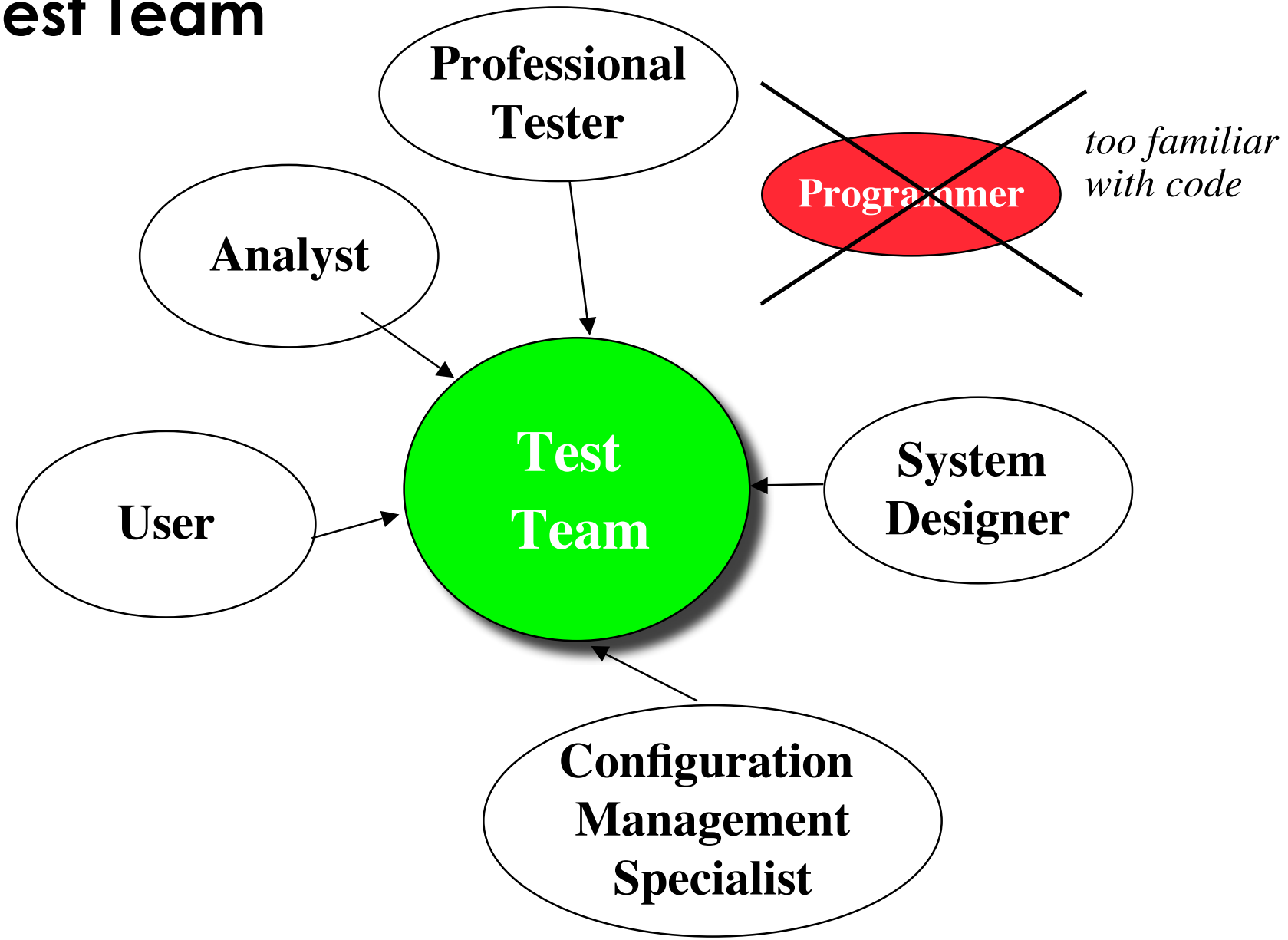
Execute the tests

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
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Design patterns can be used for integration testing
- Testing has its own lifecycle

Announcement: LS 1 Seminars in SS 2007

Applied Software Engineering, Prof. B. Bruegge, Ph.D

- Offshore Software Testing
  - http://wwwbruegge.in.tum.de/static/contribute/Lehrstuhl/OffshoreSoftwareTestingSoSe07.htm
- Knowledge Management in Software Engineering
  - http://wwwbruegge.in.tum.de.de/twiki/bin/view/Lehrstuhl/KMinSESoSe2007
- Product Line Requirements Engineering
  - http://wwwbruegge.in.tum.de/twiki/bin/view/Lehrstuhl/ProductLines
- Agile Project Management
  - http://wwwbruegge.in.tum.de/twiki/bin/view/Lehrstuhl/AgilePMSoSe2007
- **Applications are still possible!**

Additional Reading

- JUnit Website www.junit.org/index.htm
- J. Thomas, M. Young, K. Brown, A. Glover, Java Testing Patterns, Wiley, 2004
- 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.