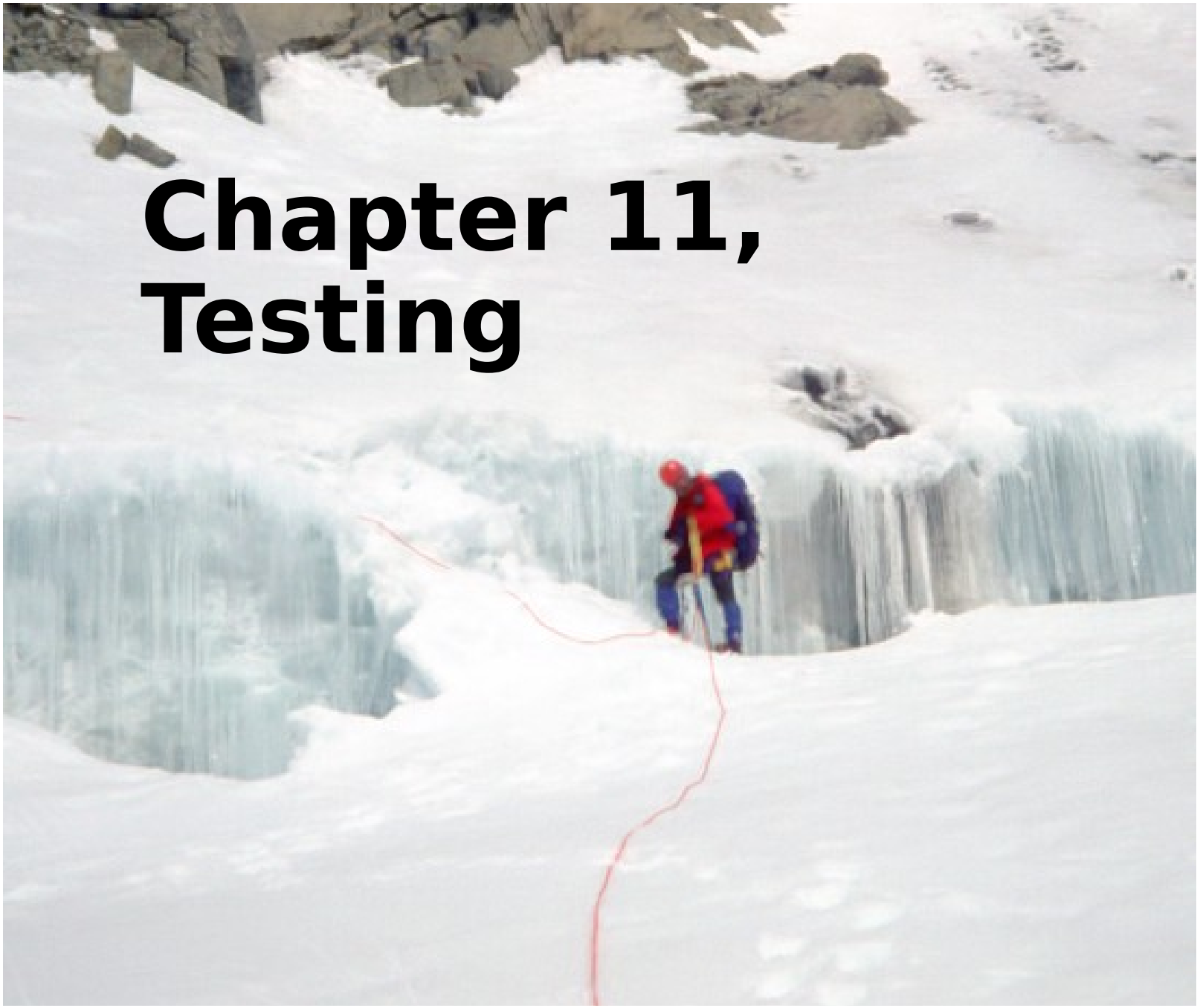
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 11, Testing



Edsger W. Dijkstra

- Testing shows the presence, not the absence of bugs
- Program testing can be used to show the presence of bugs, but never to show their absence!
- Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
- A convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box.
- Simplicity is prerequisite for reliability.

Testing – the problem

The process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system.

- Unit testing finds differences between a specification of an object and its realization as a component.
- Structural testing finds differences between the system design model and a subset of integrated subsystems.
- Functional testing finds differences between the use case model and the system.
- Performance testing finds differences between nonfunctional requirements and actual system performance.

Testing – the problem_2

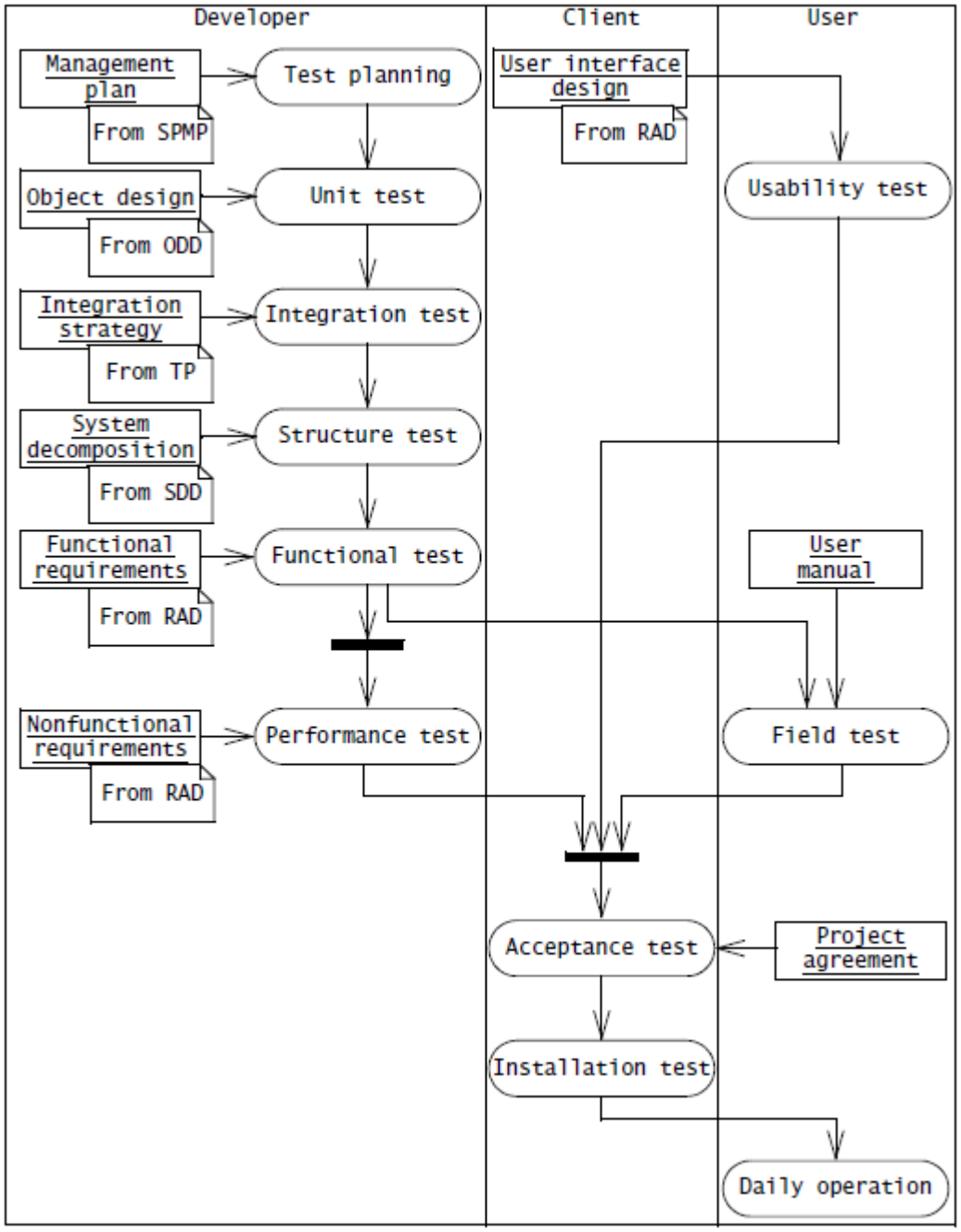
- From a modeling point of view, testing is the attempt to show that the implementation of the system is inconsistent with the system models.
- The goal of testing is to design tests that exercise defects in the system and to reveal problems.
- This activity is contrary to all other activities we described in previous chapters: analysis, design, implementation, communication, and negotiation are constructive activities.
- Testing, is aimed at breaking the system.

Testing – the problem_3

- Usually, testing is accomplished by developers that were not involved with the construction of the system.
- Unfortunately, it is impossible to completely test a nontrivial system.
  - First, testing is not decidable.
  - Second, testing must be performed under time and budget constraints.
- As a result, systems are often deployed without being completely tested, leading to faults discovered by end users.

Testing – the problem_4

- Testing is often viewed as a job that can be done by beginners. Managers would assign the new members to the testing team, because the experienced people detested testing or are needed for the more important jobs of analysis and design. Unfortunately, such an attitude leads to many problems.
- To test a system effectively, a tester must have a detailed understanding of the whole system, ranging from the requirements to system design decisions and implementation issues.
- A tester must also be knowledgeable of testing techniques and apply these techniques effectively and efficiently to meet time, budget, and quality constraints.



SPMP - **S**oftware **P**roject **M**anagement **P**lan describes all the managerial aspects of the project, in particular the work breakdown structure, the schedule, organization, work packages, and budget.

OOD - **O**bject **O**riented **D**esign

TP - Test **P**lan

Testing activities and their related work products

SDD - **S**ystem **D**esign **D**ocument

RAD - **R**equirements **A**nalysis **D**ocur

Testing concepts

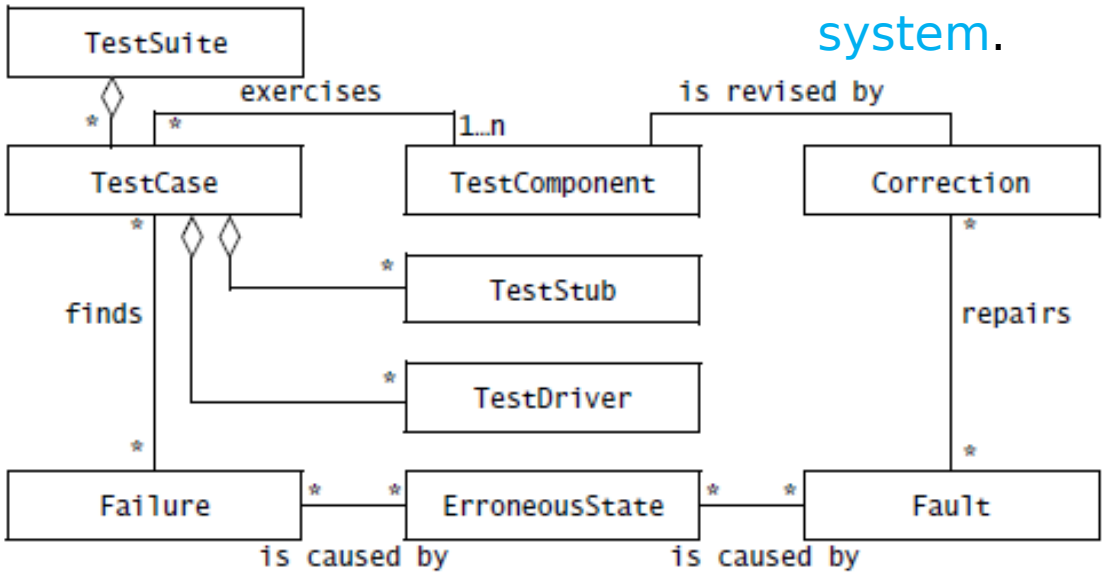
A test case is a set of input data and expected results that exercises a component with the purpose of causing failures and detecting faults. A test case has five attributes: name, location, input, oracle, and log. Test cases requires the tested component to be isolated from the rest of the system.

Table 11-1 Attributes of the class `TestCase`.

Attributes	Description
`name`	Name of test case
`location`	Full path name of executable
`input`	Input data or commands
`oracle`	Expected test results against which the output of the test is compared
`log`	Output produced by the test

Testing concepts_2

Test drivers and test stubs are used to substitute for missing parts of the system.



Model elements used during testing

A test driver simulates the part of the system that calls the component under test. A test driver passes the test inputs identified in the test case analysis to the component and displays the results.

A test stub simulates a component that is called by the tested component. The test stub must provide the same API as the method of the simulated component and must return a value compliant with the return result type of the method's type signature.

Famous Problems

- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software

- The Therac-25 accidents (1985-1987), quite possibly the most serious non-military computer-related failure ever in terms of human life (at least five died)
  - Reason: Bad event handling in the GUI
- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem.

Terminology

- Software reliability is the probability that a software system will not cause system failure for a specified time under specified conditions [IEEE Std. 982.2-1988].
- Failure: Any deviation of the observed behavior from the specified behavior
- Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure
- Fault: The mechanical or algorithmic cause of an error ("bug")

Terminology_2

- The goal of testing is to maximize the number of discovered faults, which then allows developers to correct them and increase the reliability of the system.
- We define testing as the systematic attempt to find faults in a planned way in the implemented software.
- Contrast this definition with another common one: "*testing is the process of demonstrating* that *faults are not present*."
- *The distinction between these two definitions is important*.

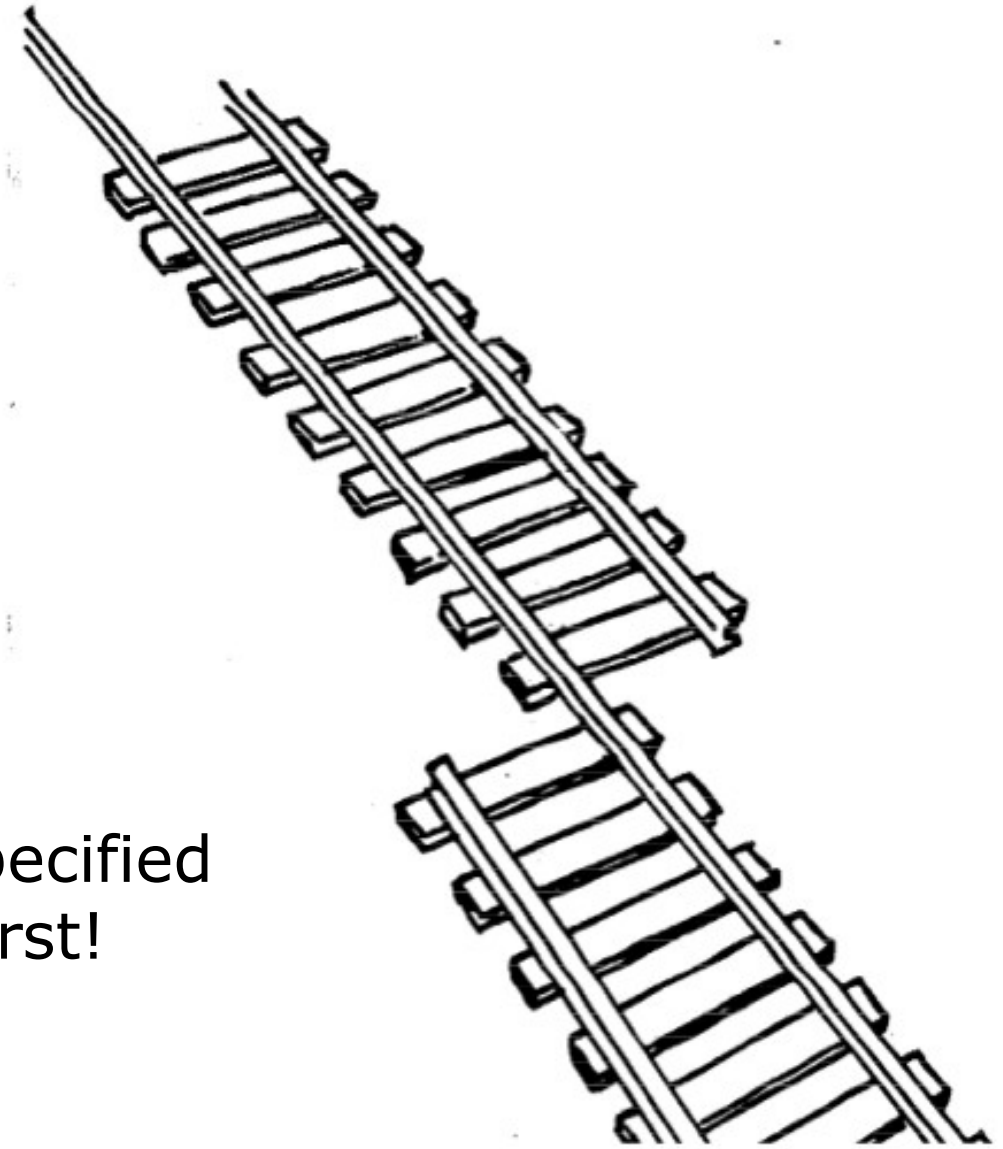
Terminology_3

- Our definition does not mean that we simply demonstrate that the program does what it is intended to do.
- Validation: Activity of checking for deviations between the observed behavior of a system and its specification.
- Testing requires a different thinking, in that developers try to detect faults in the system, that is, differences between the reality of the system and the requirements. Many developers find this difficult to do. One reason is the way we use the word "success" during testing.

Terminology_4

- Many project managers call a test case "successful" if it does not find a fault; that is, they use the second definition of testing during development.
- However, because "successful" denotes an achievement, and "unsuccessful" means something undesirable, these words should not be used in this fashion during testing.

What is this?



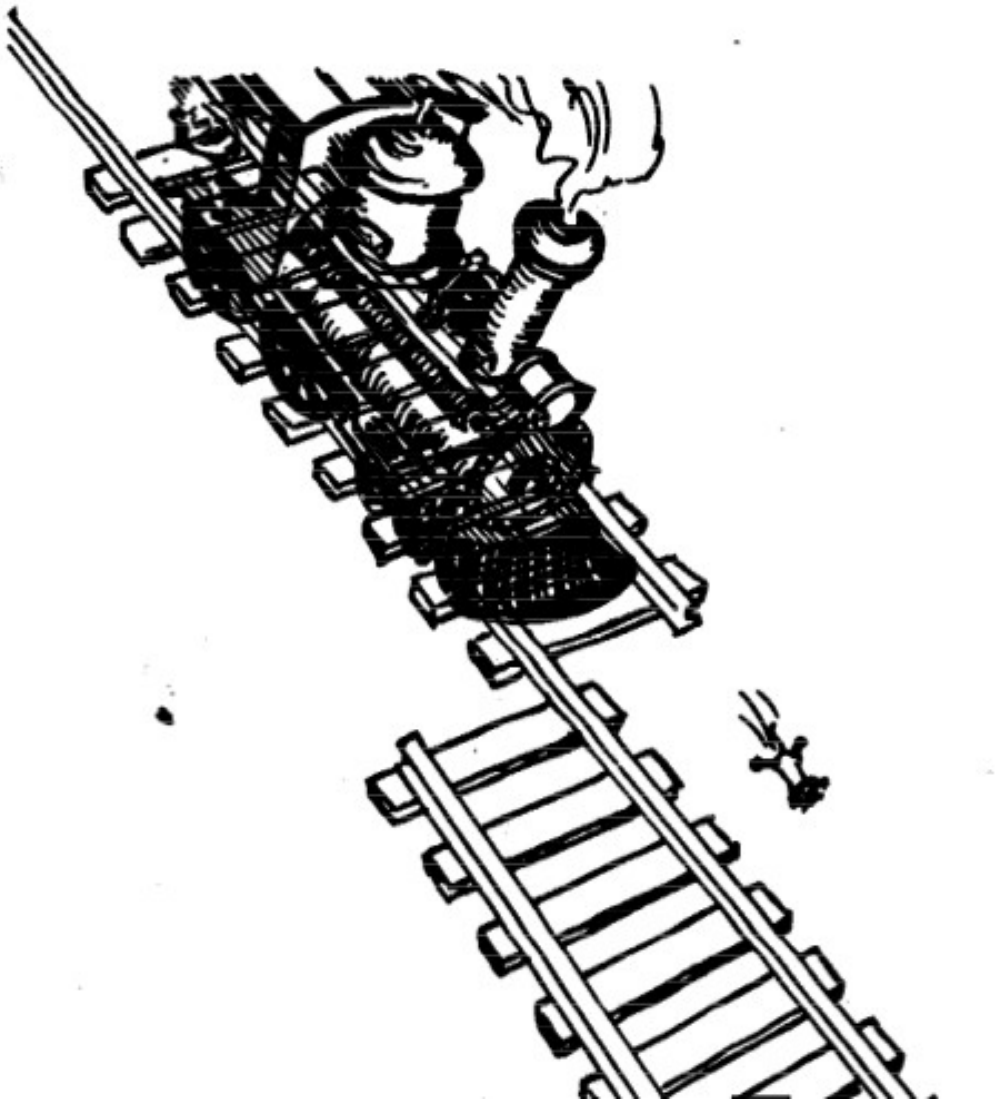
A failure?

An error?

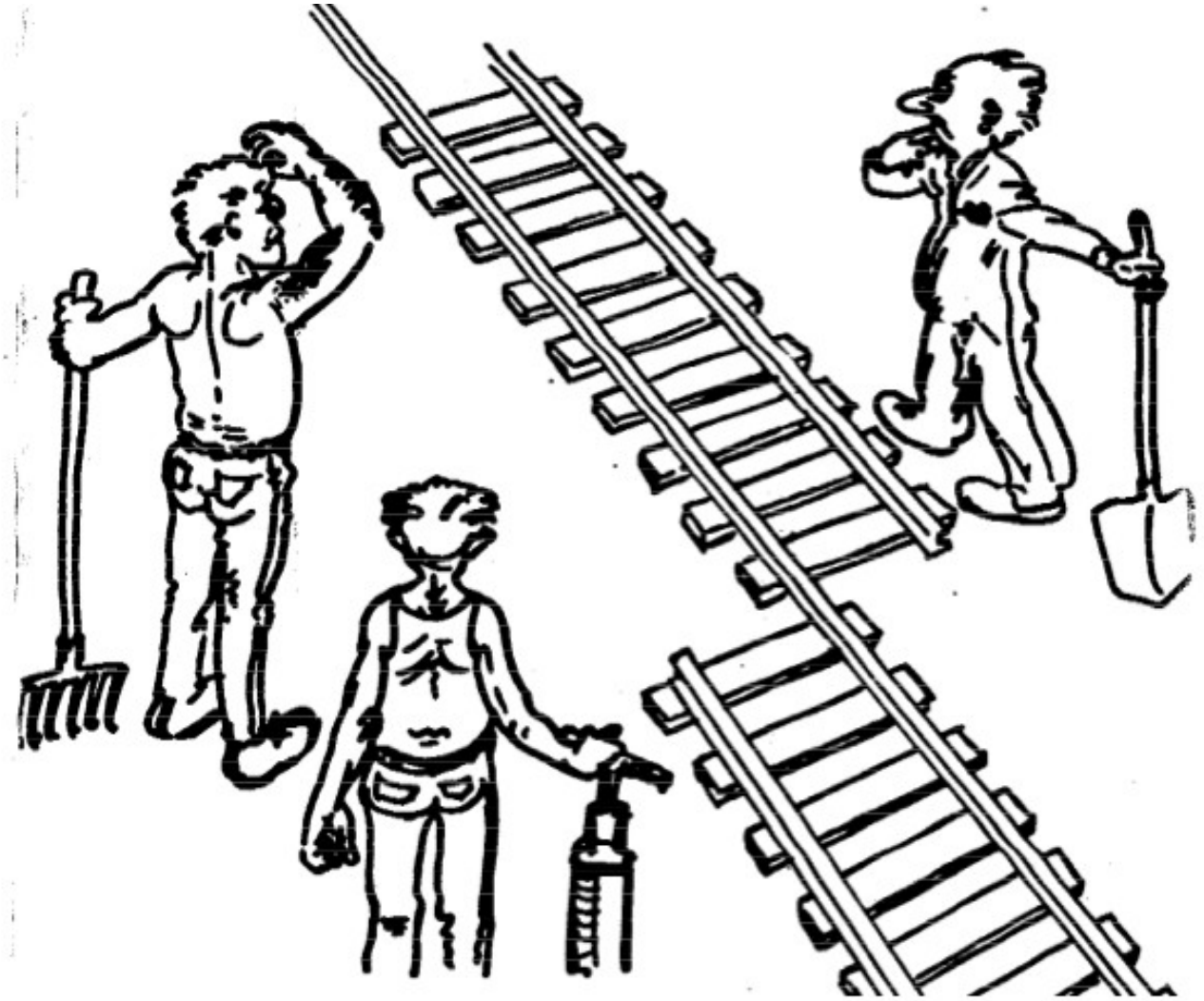
A fault?

We need to describe specified and desired behavior first!

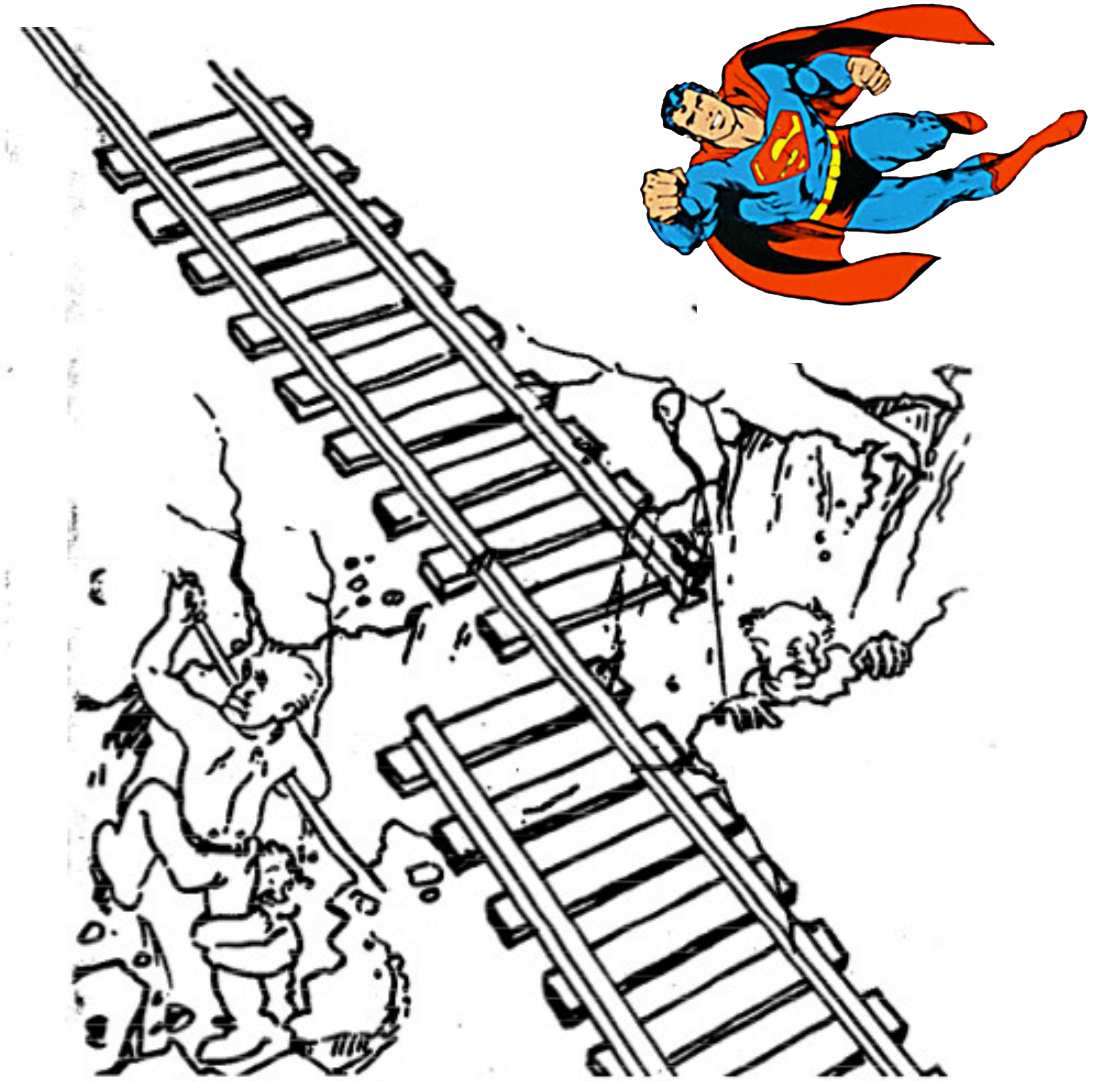
Erroneous State (“Error”)



Algorithmic Fault



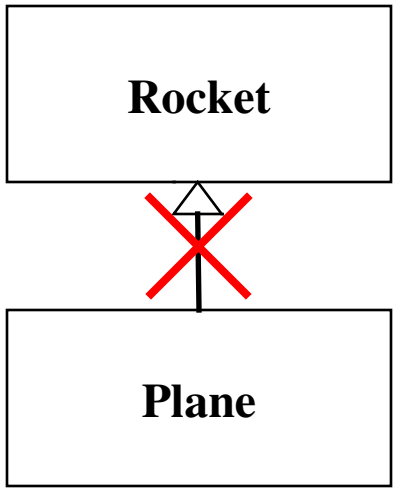
Mechanical Fault



F-16 Bug



- What is the failure?
- What is the error?
- What is the fault?
  - Bad use of implementation inheritance
  - A Plane is **not** a rocket.



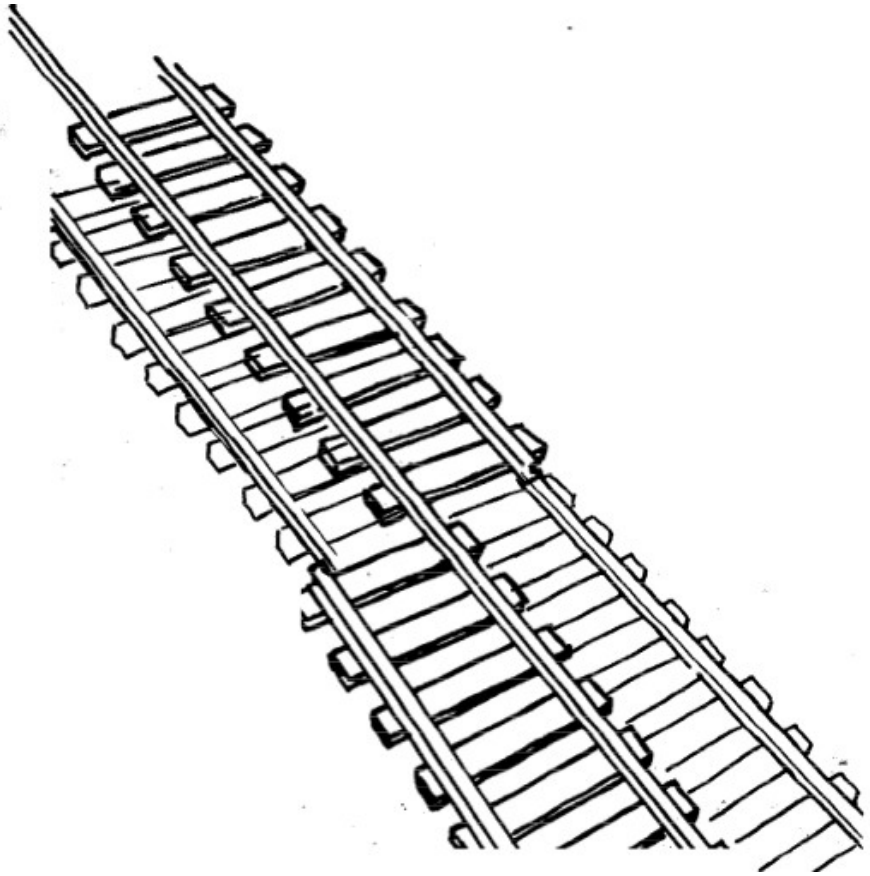
Examples of Faults and Errors

- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null

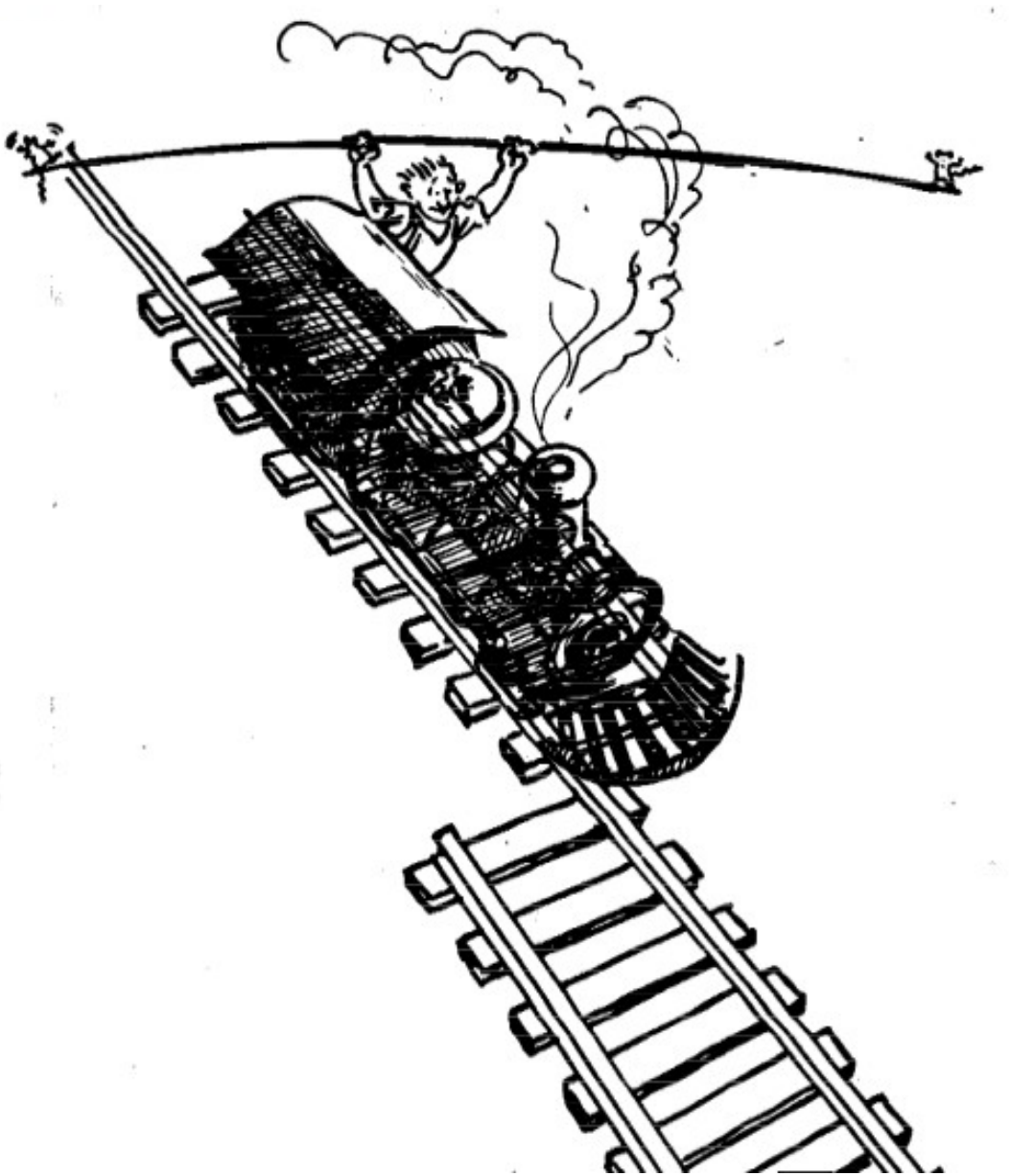
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Null reference errors
  - Concurrency errors
  - Exceptions.

How do we deal with Errors, Failures and Faults?

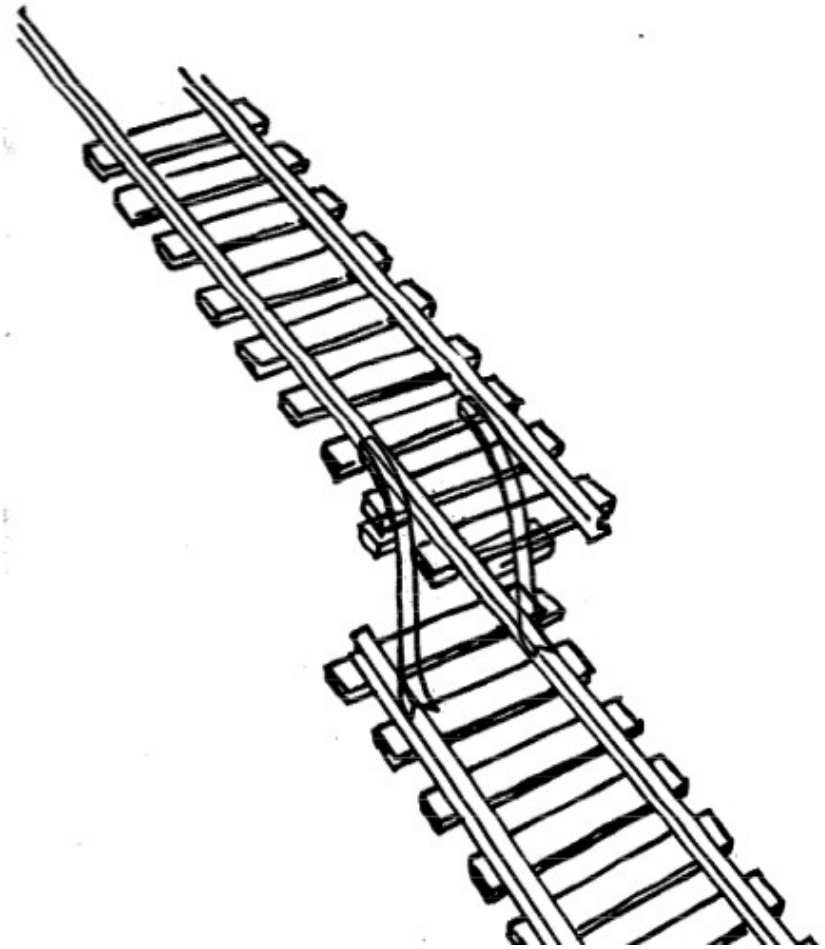
Modular Redundancy



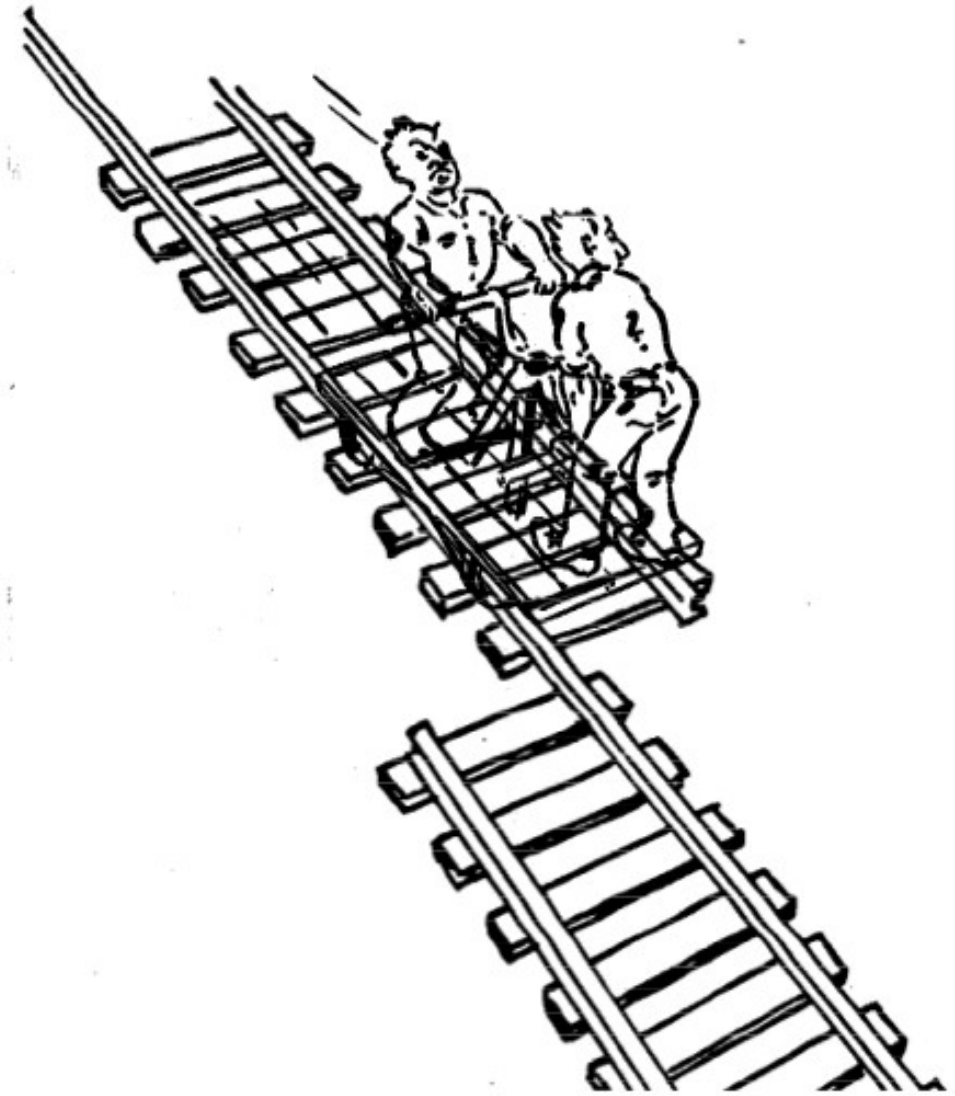
Declaring the Bug as a Feature



Patching



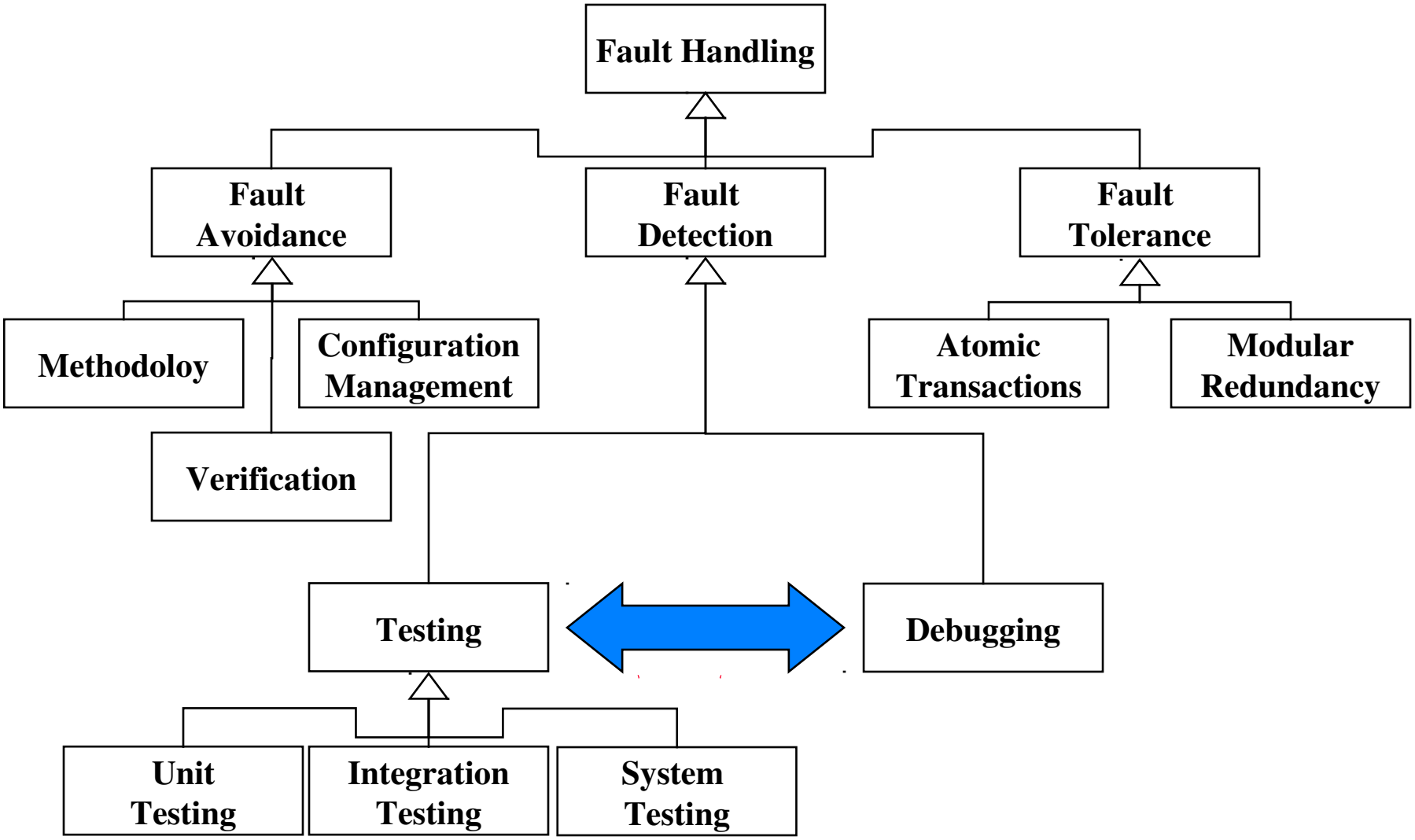
Testing



Another View on How to Deal with Faults

- Fault avoidance
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (Faults) of an observed failure
  - Monitoring: Deliver information about state => Used during debugging
- Fault tolerance
  - Exception handling
  - Modular redundancy.

Taxonomy for Fault Handling Techniques



Observations

- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- Testing is not for free

=> Define your goals and priorities

Component inspection

- Inspections find faults in a component by reviewing its source code in a formal meeting. Inspections can be conducted before or after the unit test.

- The inspection is conducted by a team of developers, including the author of the component, a moderator who facilitates the process, and one or more reviewers who find faults in the component.

- Fagan's inspection method [Fagan, 1976]. consists of five steps:

  1. Overview. The author of the component briefly presents the purpose and scope of the component and the goals of the inspection.

Component inspection_2

2. Preparation. The reviewers become familiar with the implementation of the component.

3. Inspection meeting. A reader paraphrases the source code of the component, and the inspection team raises issues with the component. A moderator keeps the meeting on track.

4. Rework. The author revises the component.

5. Follow-up. The moderator checks the quality of the rework and may determine the component that needs to be reinspected.

- Fagan's inspections are usually perceived as time-consuming because of the length of the preparation and inspection meeting phase. The effectiveness of a review also depends on the preparation of the reviewers.

Usability Testing

- Usability testing is focused on the user's understanding of the system. Usability testing does not compare the system against a specification. It focuses on finding differences between the system and the users' expectation of what it should do.
- Usability tests are also concerned with user interface details, such as the look and feel of the user interface, the geometrical layout of the screens, sequence of interactions, and the hardware.
- There are three types of usability tests:

Usability Testing_2

1. Scenario test. One or more users are presented with a visionary scenario of the system. Developers identify how quickly users are able to understand the scenario, how accurately it represents their model of work, and how positively they react to the description of the new system.

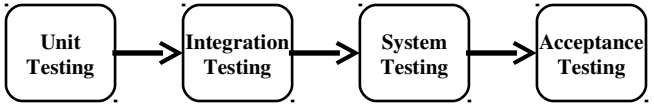
2. Prototype test. The end users are presented with a piece of software that implements key aspects of the system. A vertical prototype completely implements a use case through the system. A horizontal prototype implements a single layer in the system; an example is a user interface prototype, which presents an interface for most use cases.

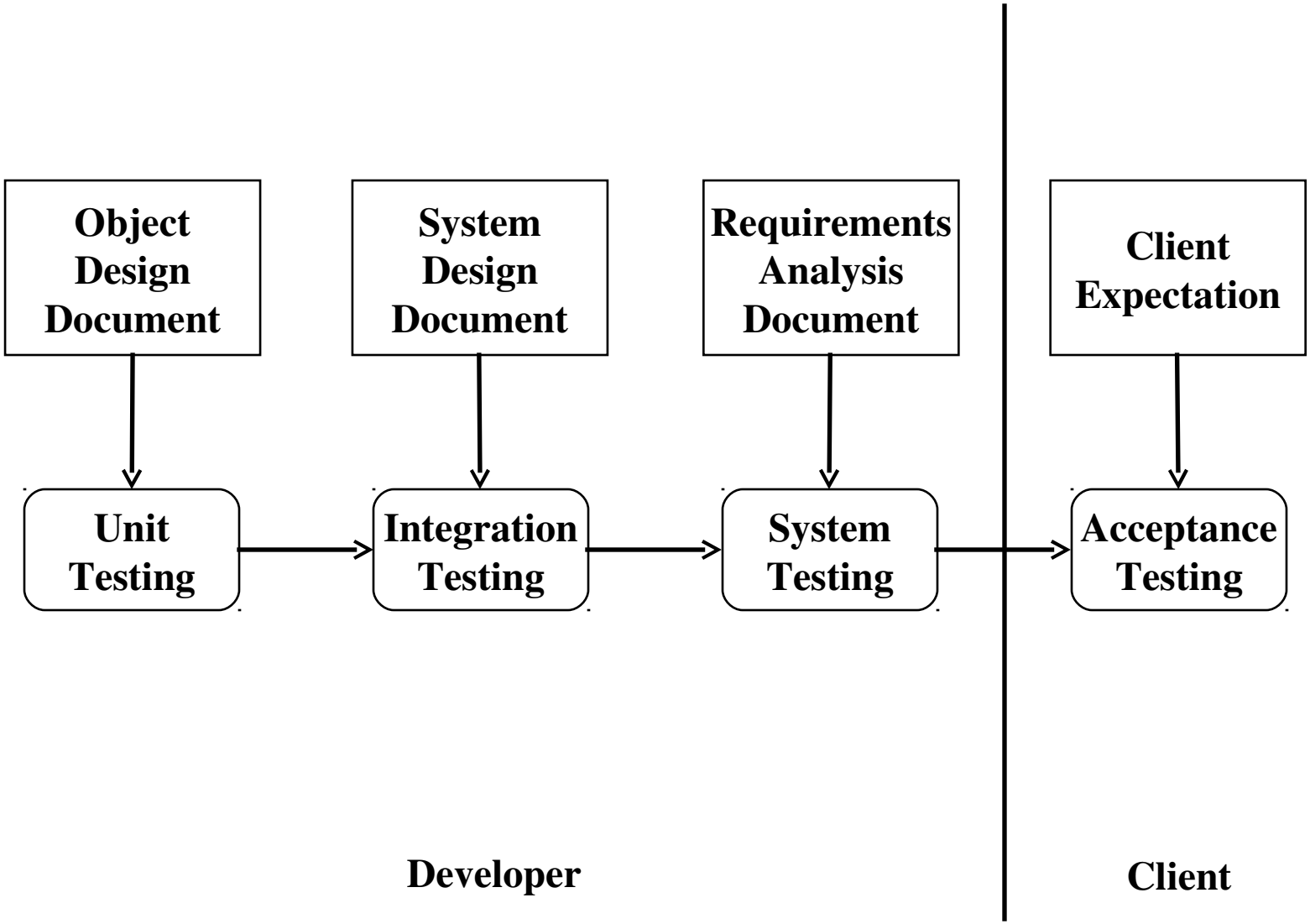
3. Product test. Similar to the prototype test except that a functional version of the system is used in place of the prototype. A product test can only be conducted after most of the system is developed. It also requires that the system be easily modifiable.

Testing takes creativity

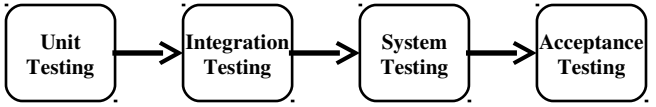
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

Testing Activities



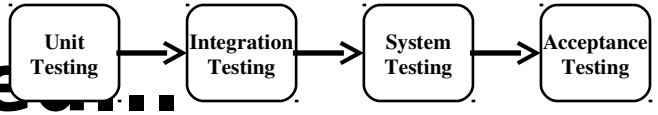


Types of Testing



- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.

Types of Testing continued...



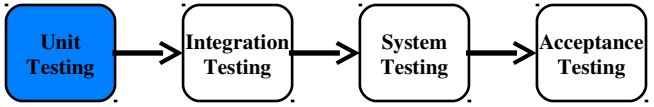
- System Testing
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

When should you write a test?

- Traditionally after the source code is written

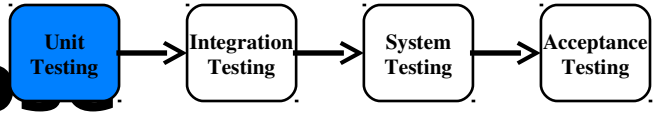
- In XP before the source code written
  - Test-Driven Development Cycle
    - Add a test
    - Run the automated tests
      => see the new one fail
    - Write some code
    - Run the automated tests
      => see them succeed
    - Refactor code.

Unit Testing

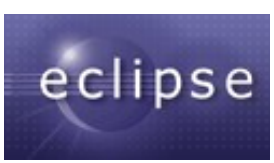


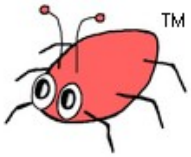
- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)
- Dynamic Testing (at run time)
  - Black-box testing
  - White-box testing.

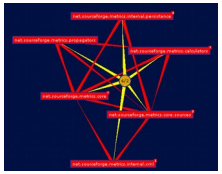
Static Analysis with Eclipse



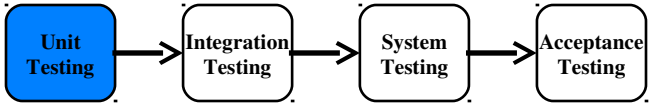
- Compiler Warnings and Errors
  - *Possibly uninitialized Variable*
  - *Undocumented empty block*
  - *Assignment has no effect*
- Checkstyle
  - Check for code guideline violations
  - http://checkstyle.sourceforge.net
- FindBugs
  - Check for code anomalies
  - http://findbugs.sourceforge.net
- Metrics
  - Check for structural anomalies
  - http://metrics.sourceforge.net







Black-box testing



- Focus: I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

Black-box testing: Test case selection

a) Input is valid across range of values
- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set
- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values

- No rules, only guidelines.

Black box testing: An example

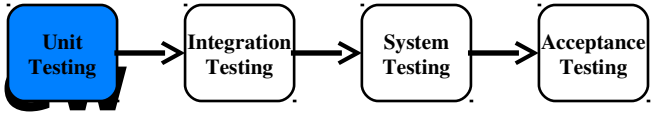
```
public class MyCalendar {

   public int getNumDaysInMonth(int month, int year)
         throws InvalidMonthException
   { ... }
}
```

Representation for month:
1: January, 2: February, …., 12: December

Representation for year:
1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of getNumDaysInMonth()?

White-box testing overview



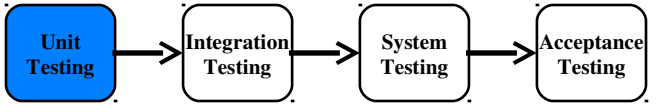
- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

=> Details in the exercise session about testing

Unit Testing Heuristics

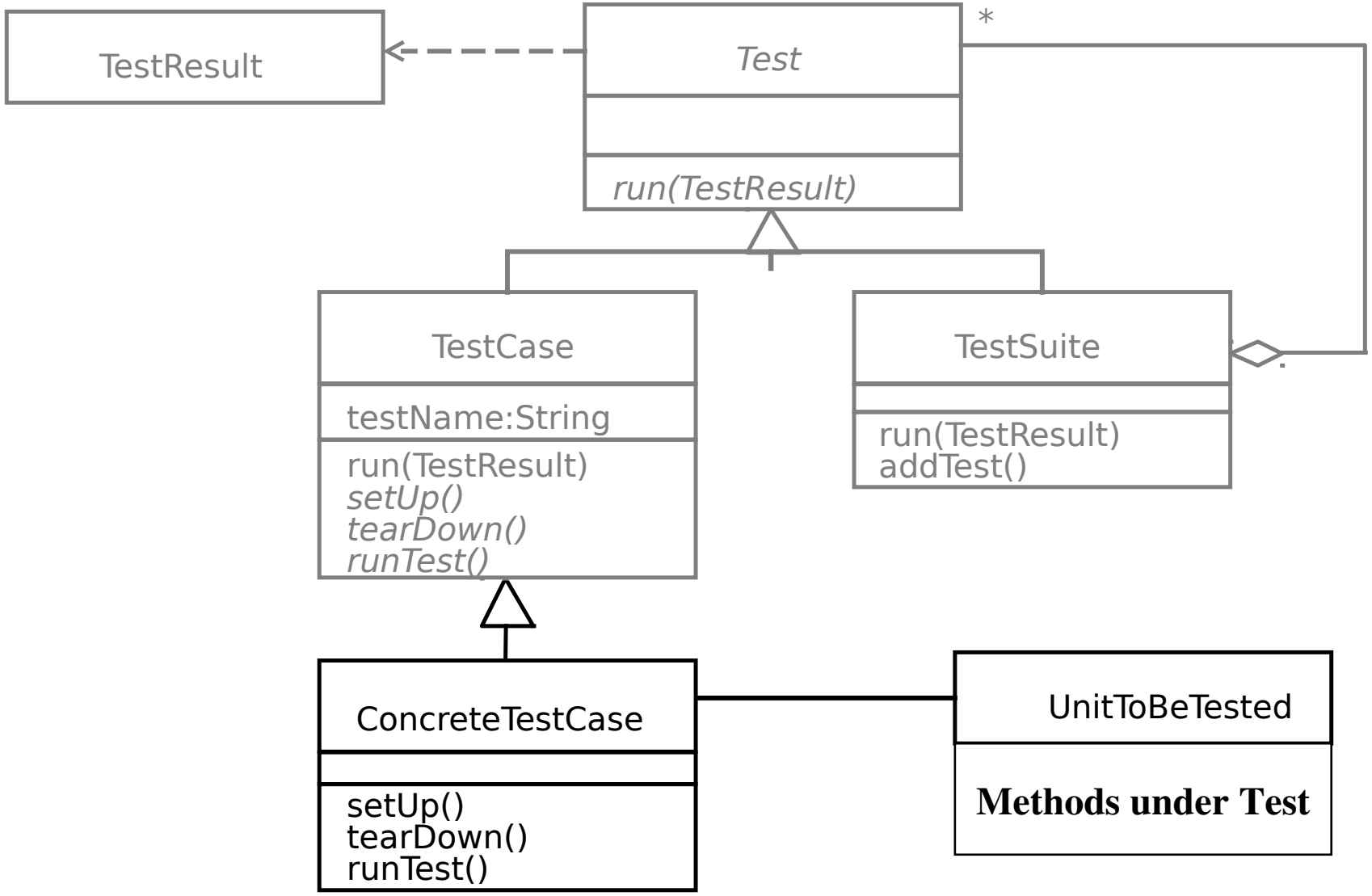
1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
2. Develop the test cases
   - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Sometimes reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
   - Automate this if possible.

JUnit: Overview



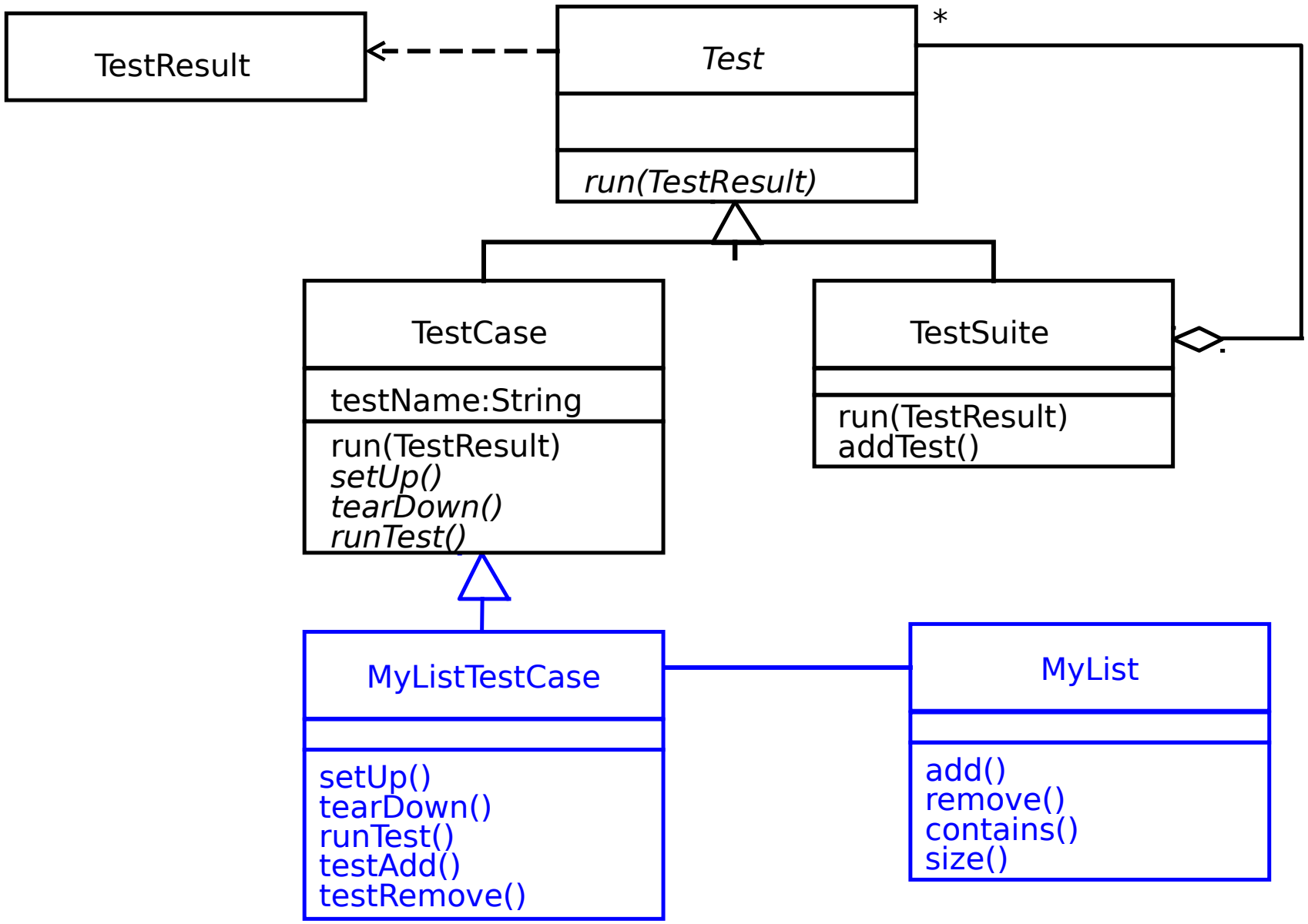
- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Written by Kent Beck and Erich Gamma
- Written with "test first" and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring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

JUnit Classes



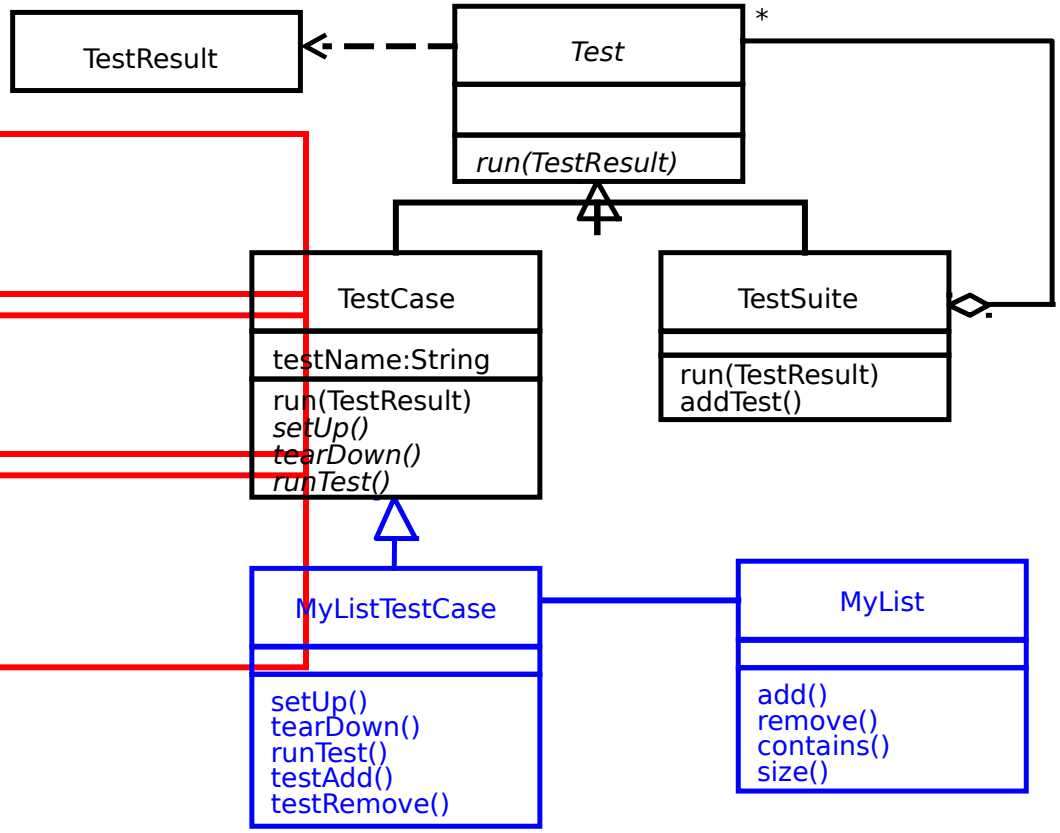
An example: Testing MyList

- 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

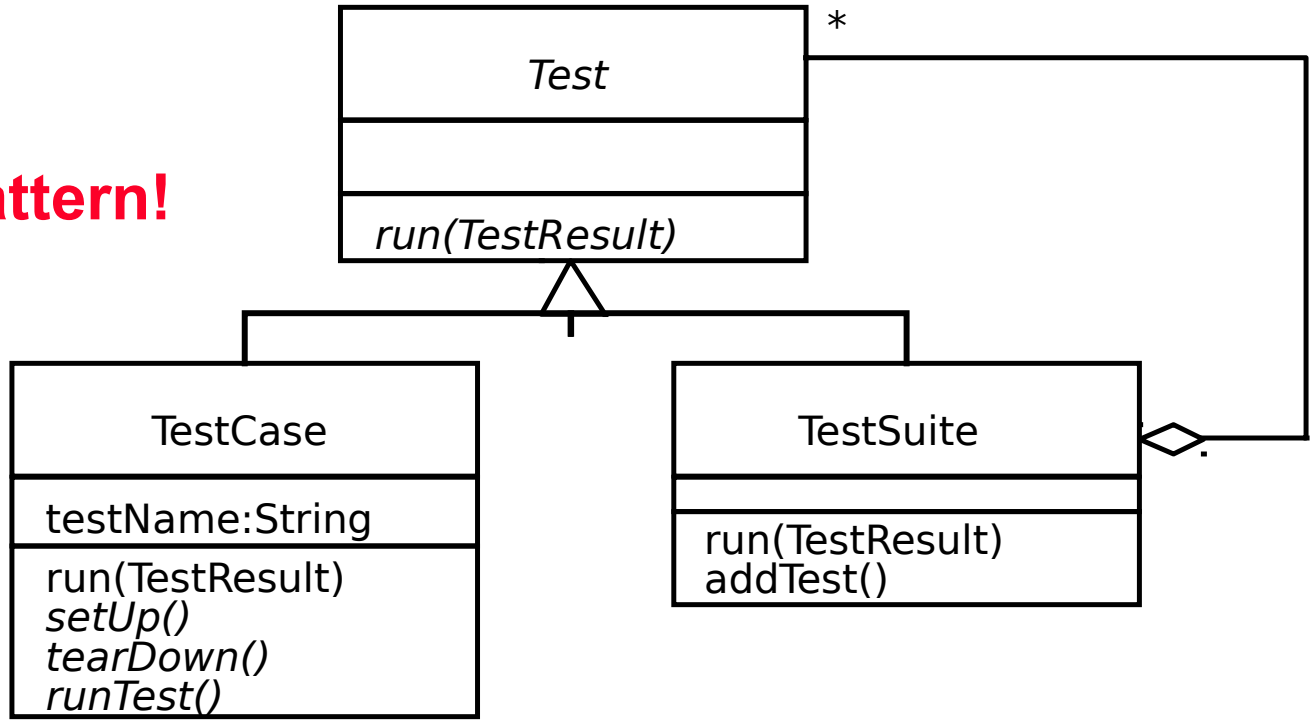
Test Case

Test Case

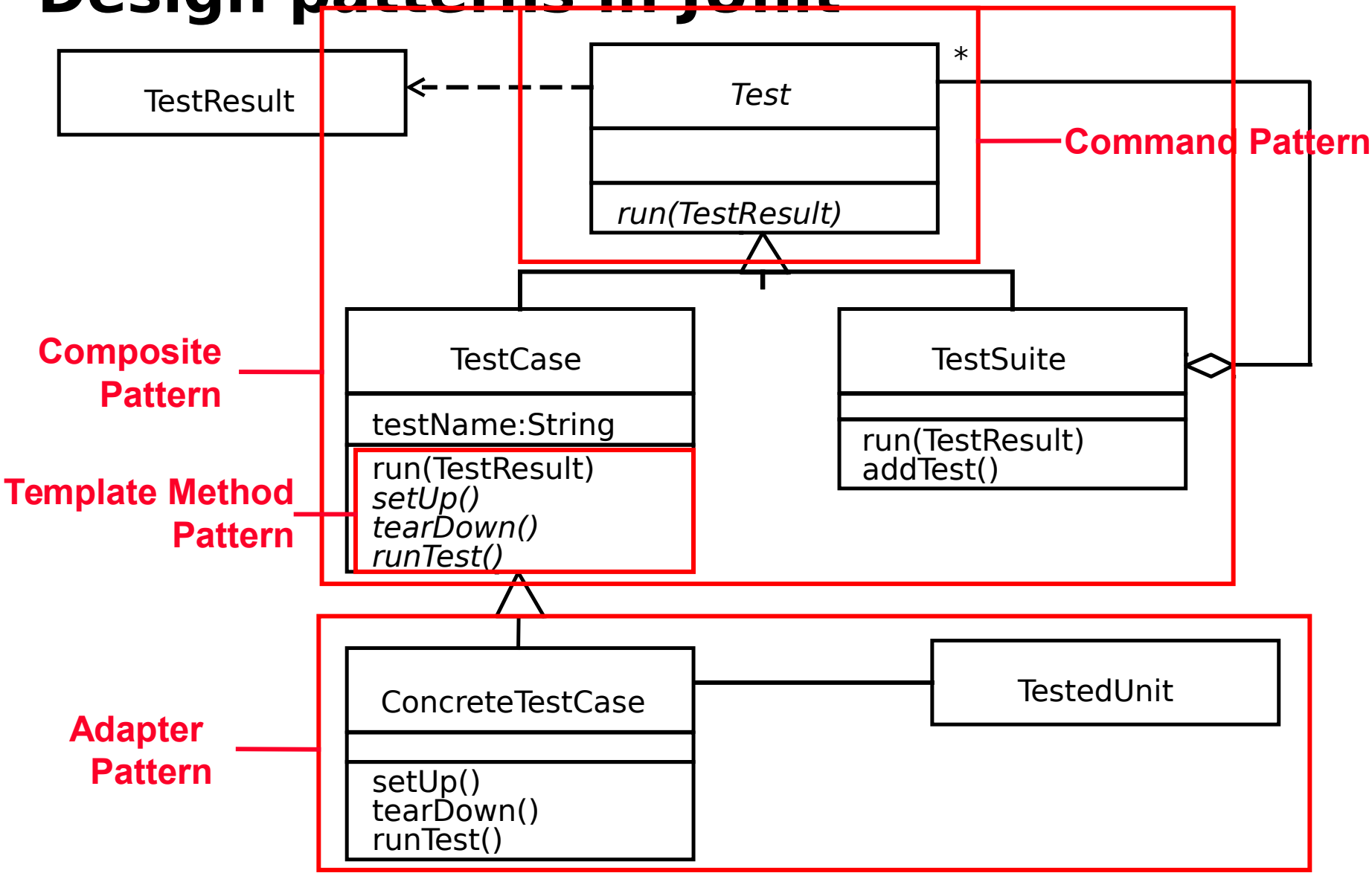
Collecting TestCases into TestSuites

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

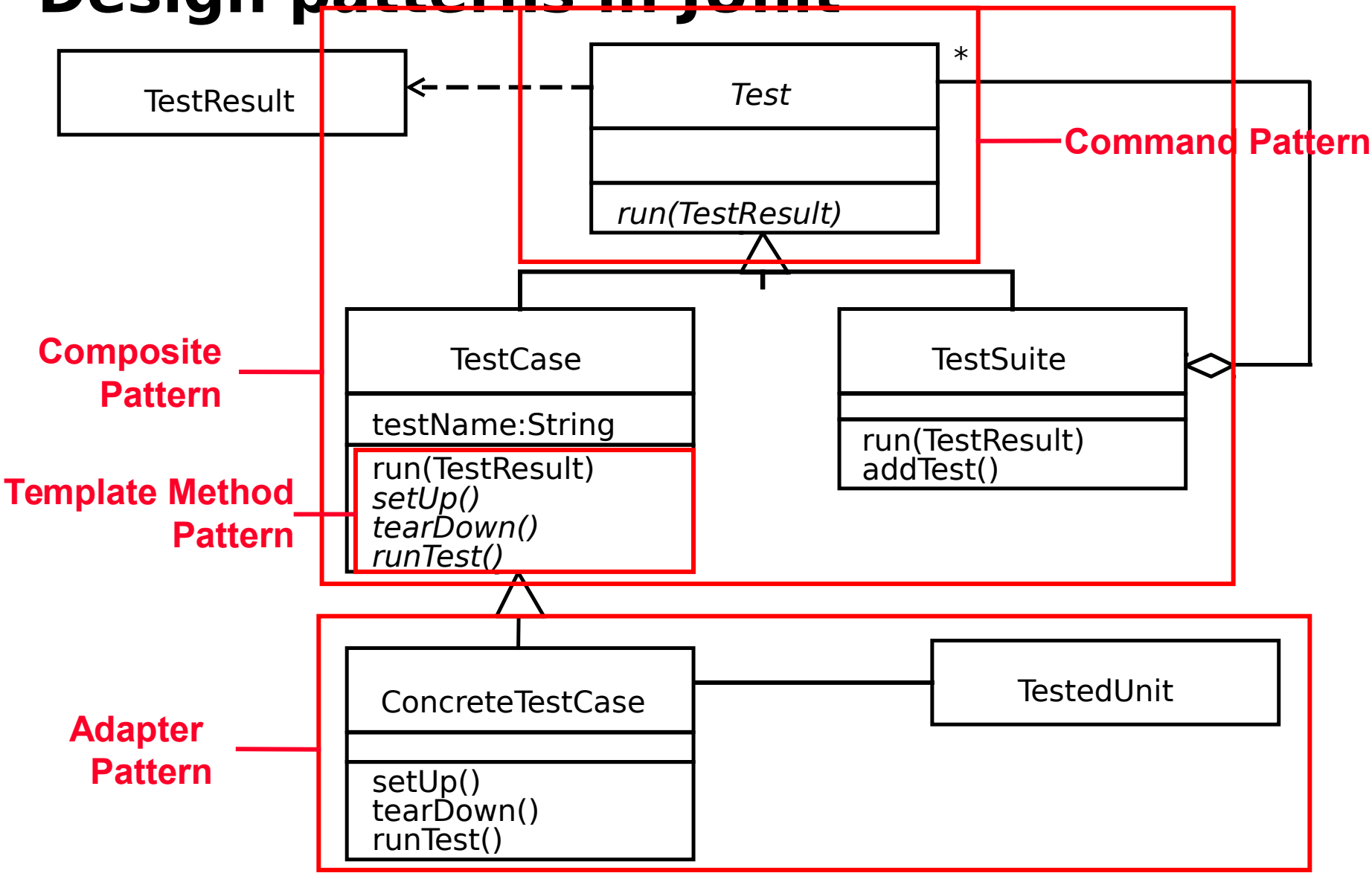
Composite Pattern!



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
  - http://maven.apache.org
    - Build and Release Management Tool
  - http://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

Additional Readings

- JUnit Website www.junit.org/index.htm