Testing I

Software Engineering I
Lecture 13

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Outline

This lecture

- Terminology
- Testing Activities
- Unit testing

Next lecture

- Integration testing
  - Testing strategy
  - Design patterns & testing
- System testing
  - Function testing
  - Acceptance testing.

Famous bugs

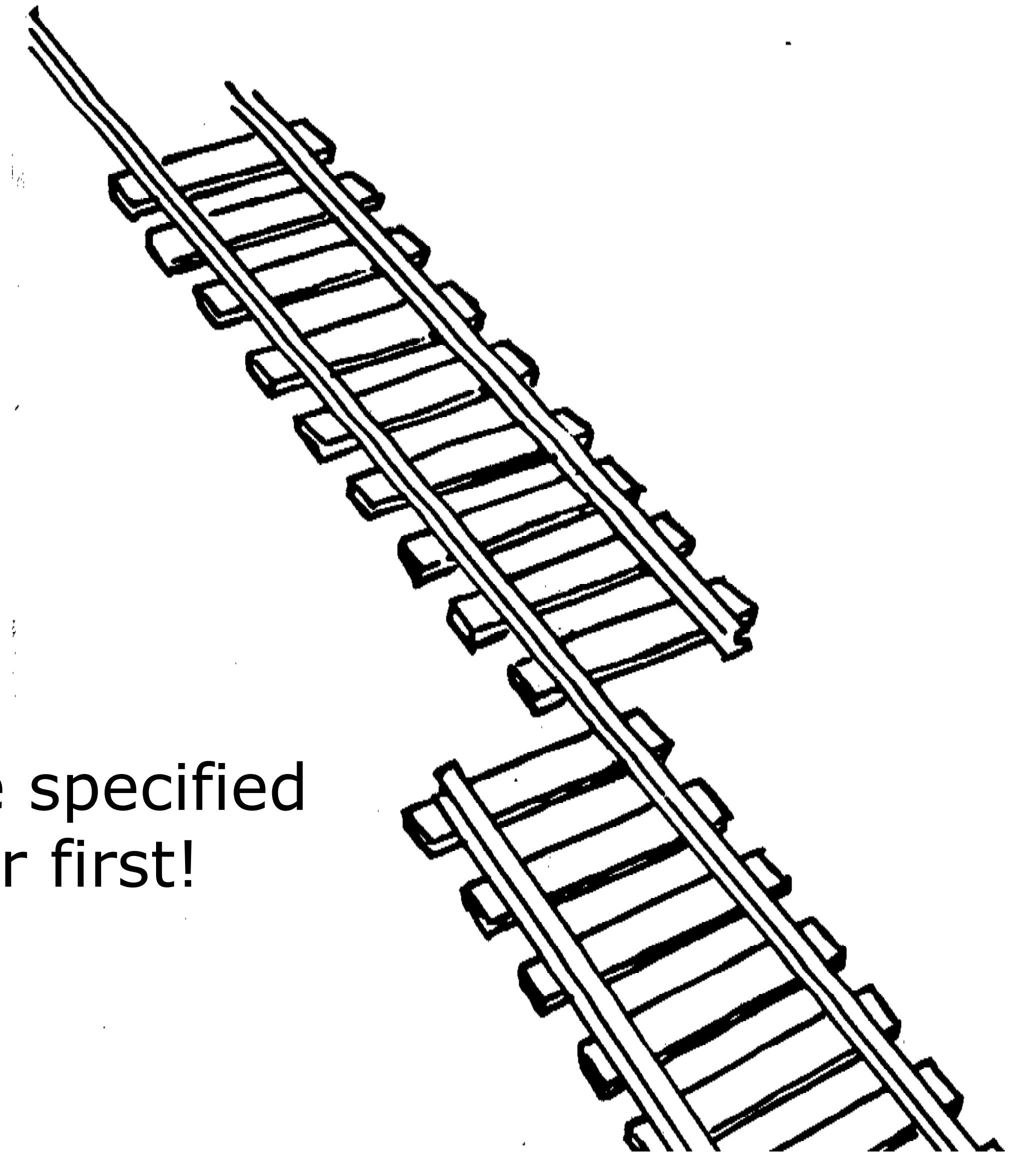
- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software

- 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

- 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")
- Validation: Activity of checking for deviations between the observed behavior of a system and its specification.

What is this?



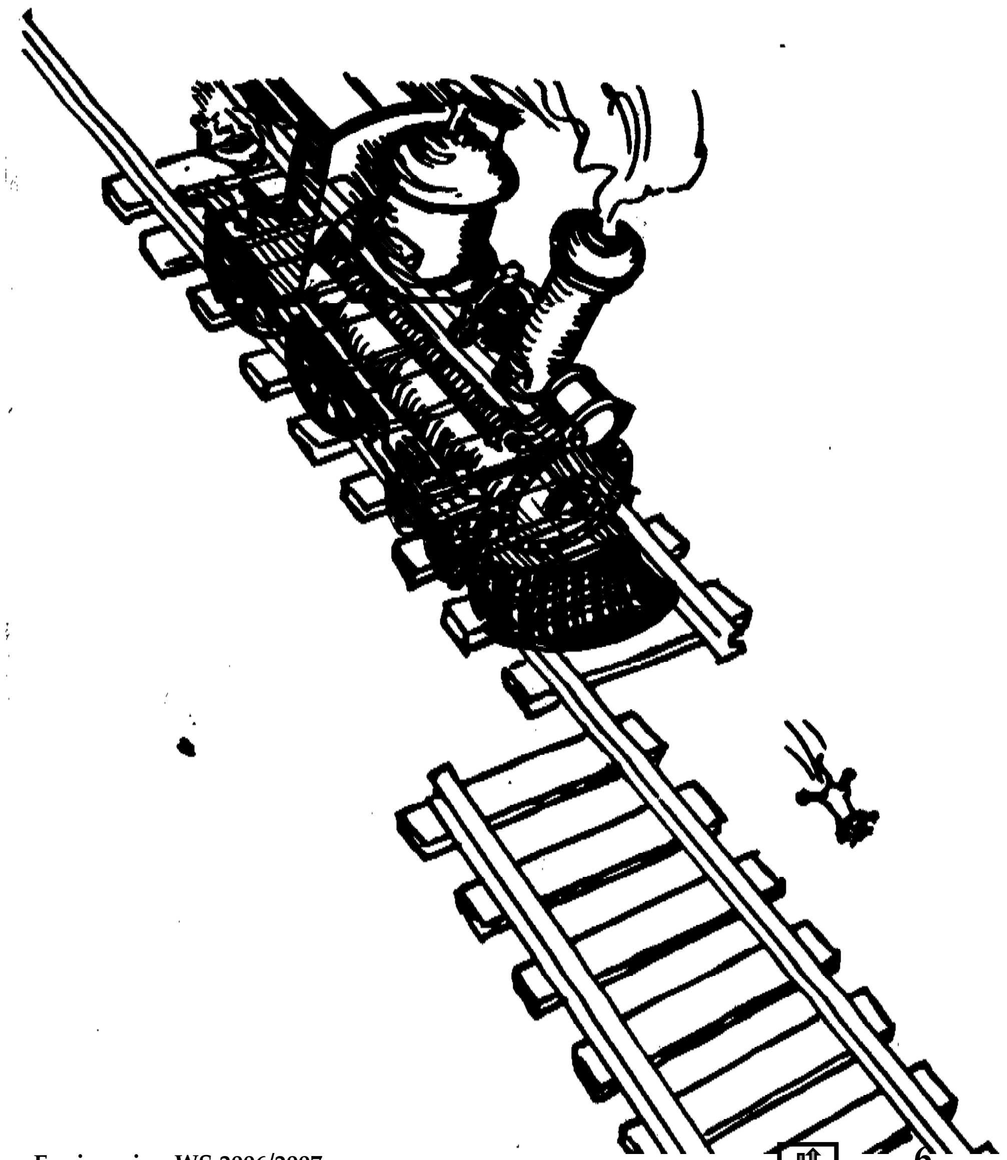
A failure?

An error?

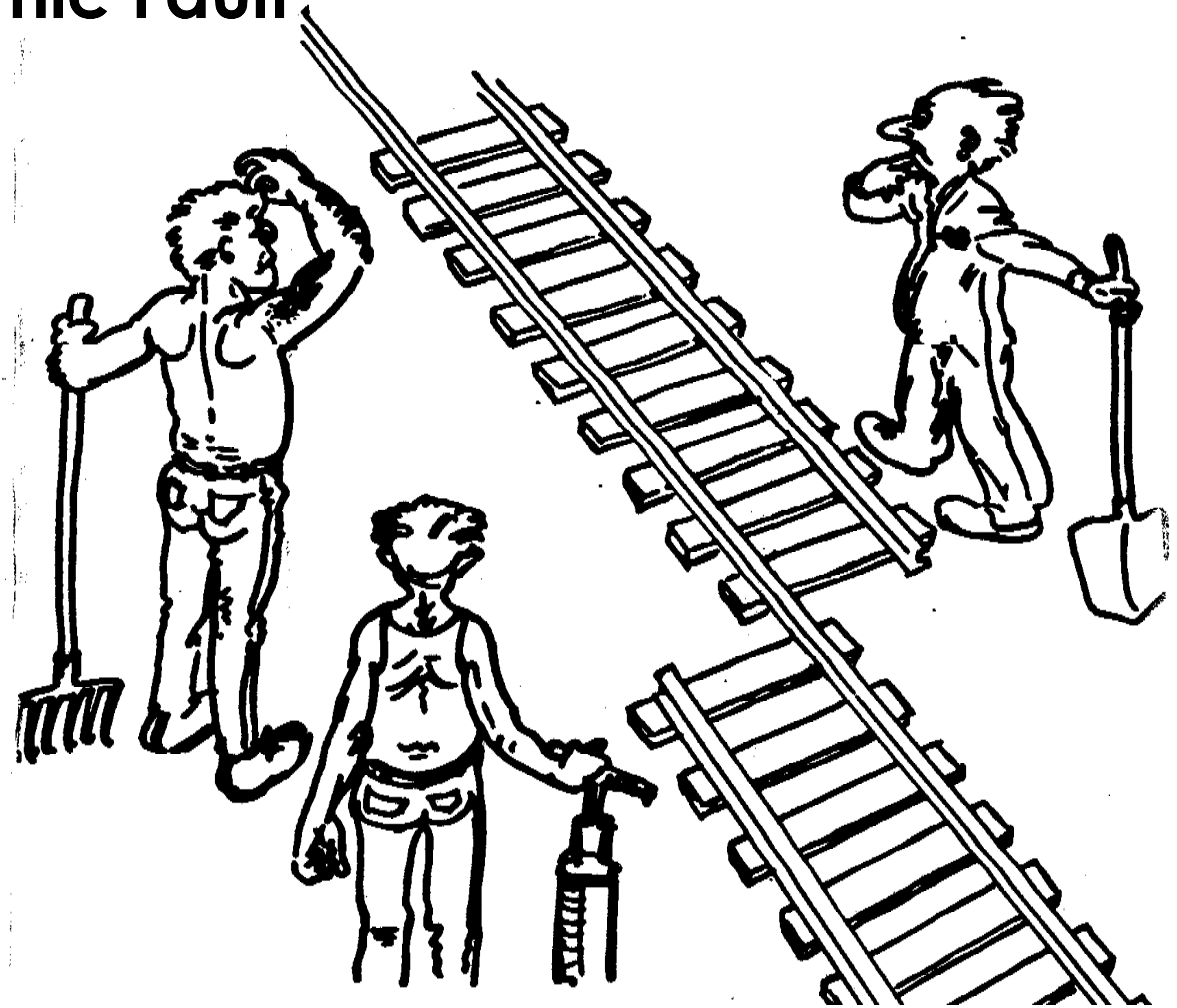
A fault?

We need to describe specified and desired behavior first!

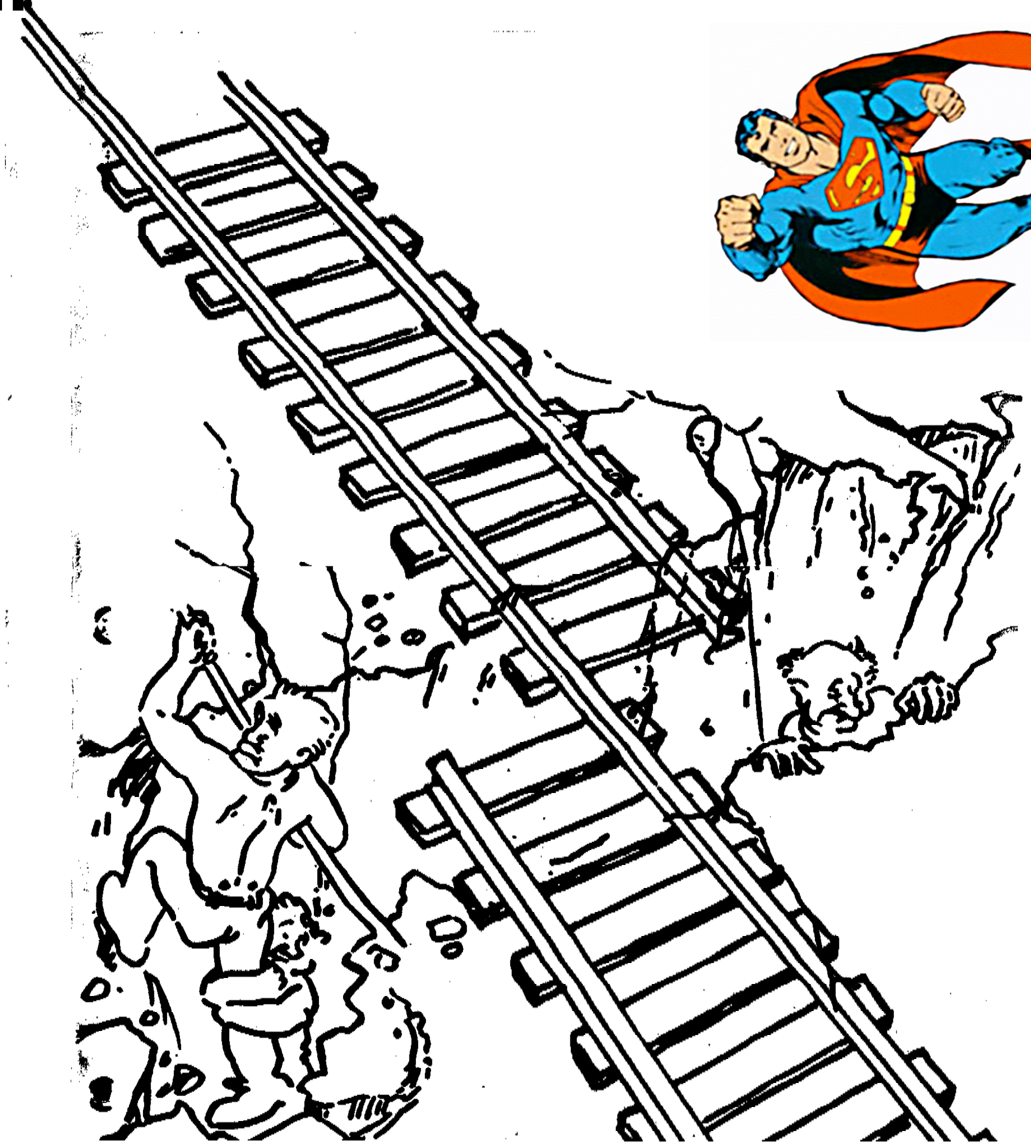
Erroneous State ("Error")



Algorithmic Fault

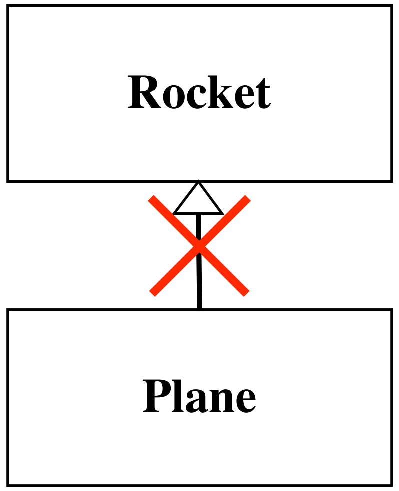


Mechanical Fault



F-16 Bug

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



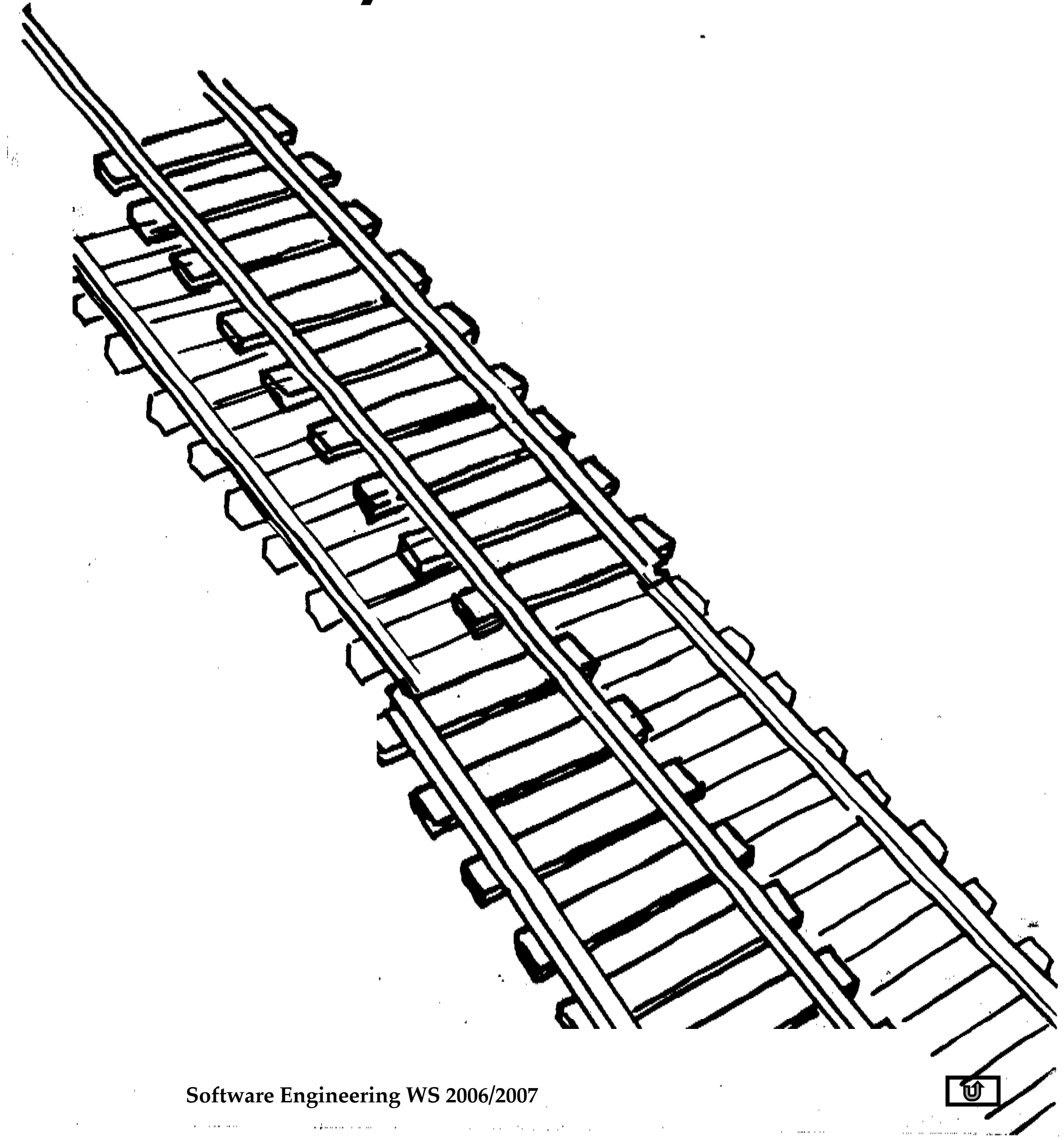
Examples of Faults and Errors

- 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

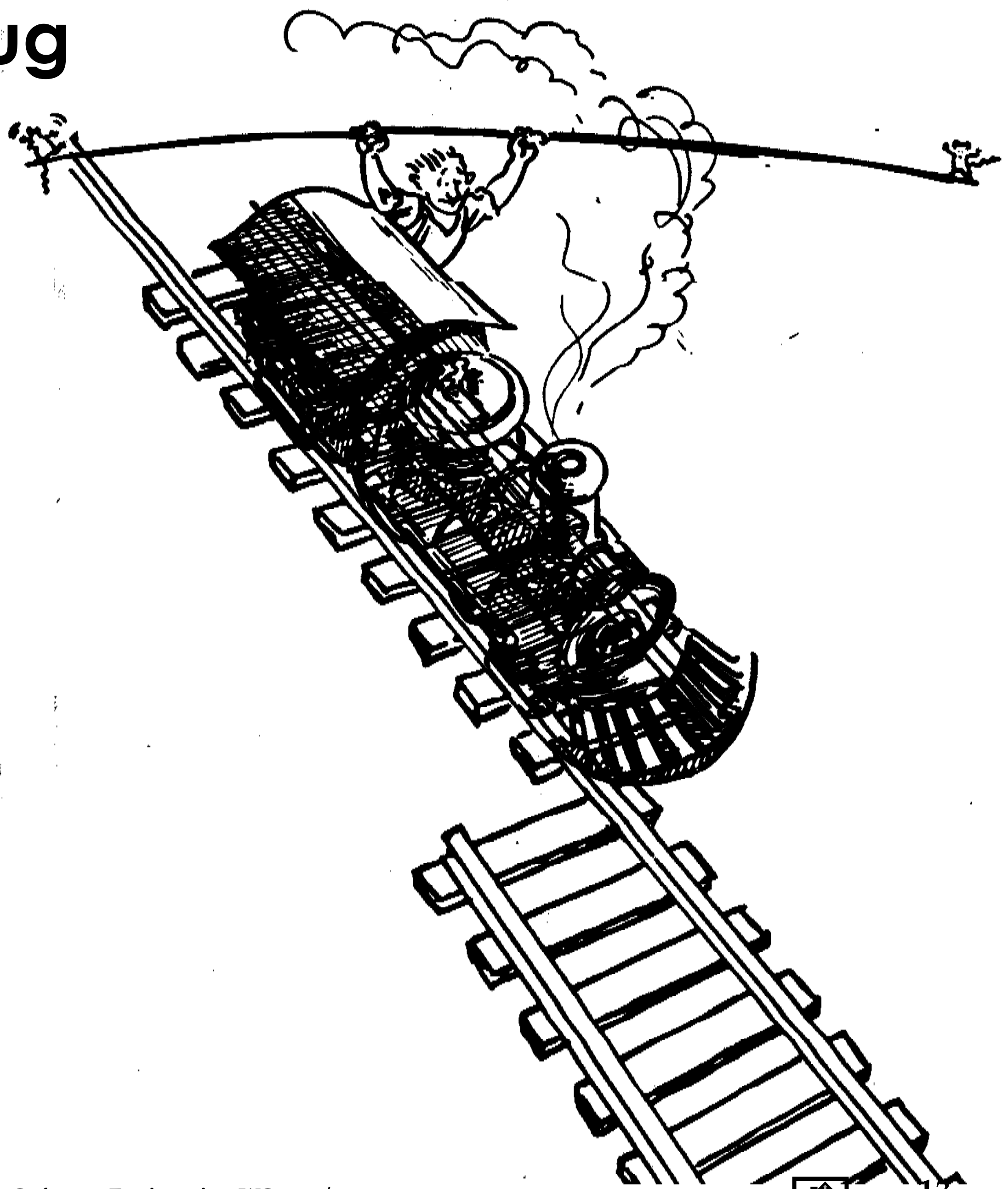
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Null reference errors
  - Concurrency errors
  - Exceptions.

How do we deal with Errors, Failures and Faults?

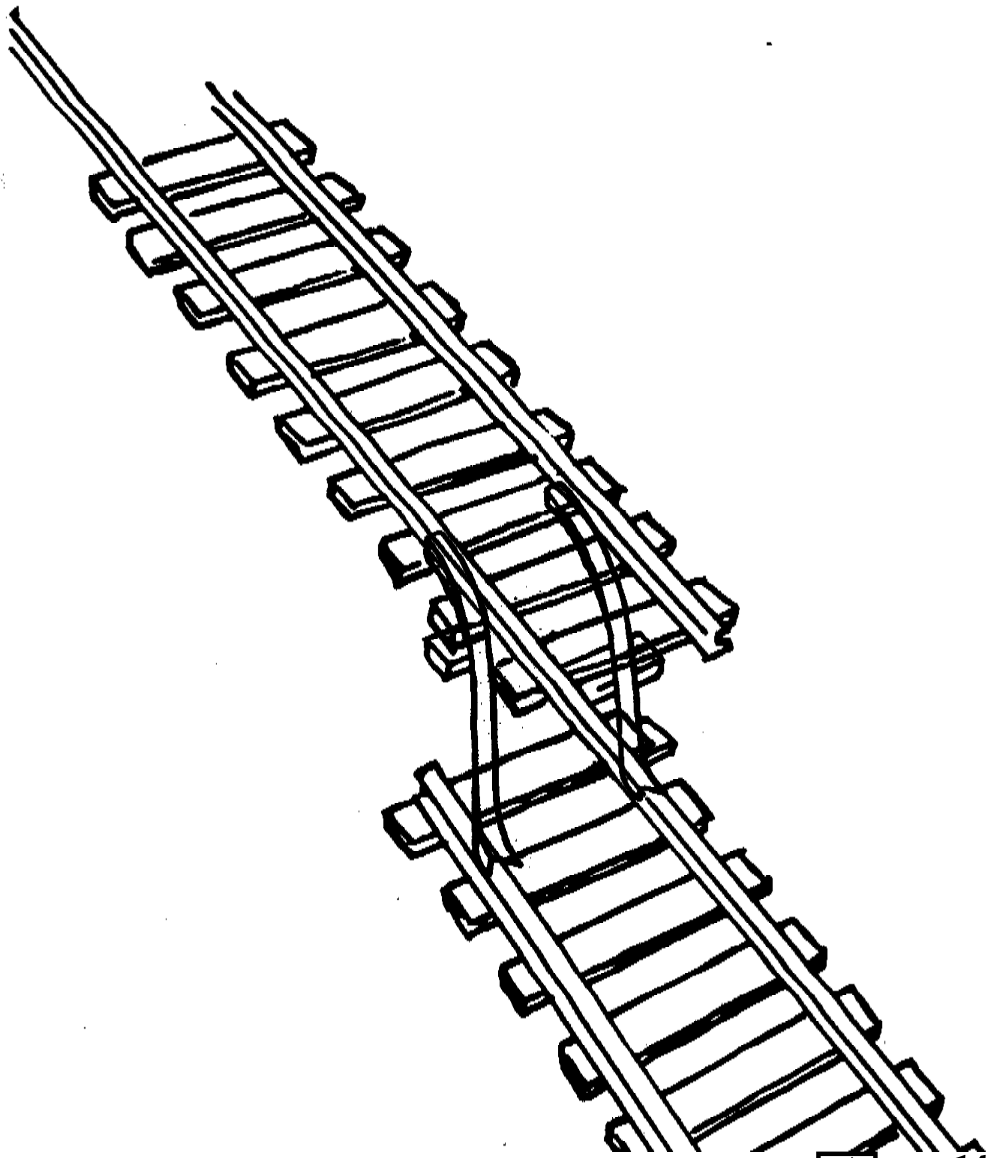
Modular Redundancy



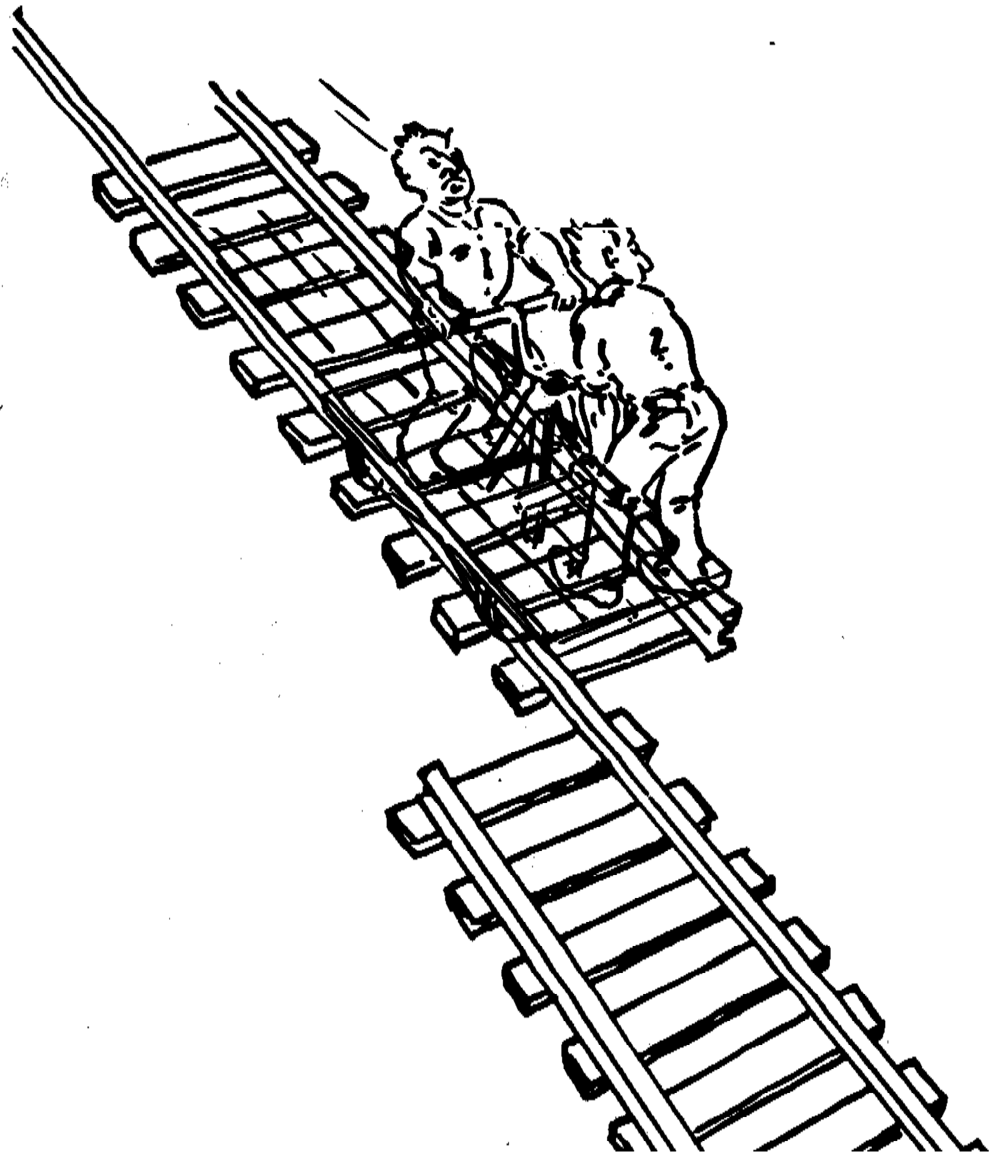
Declaring the Bug as a Feature



Patching



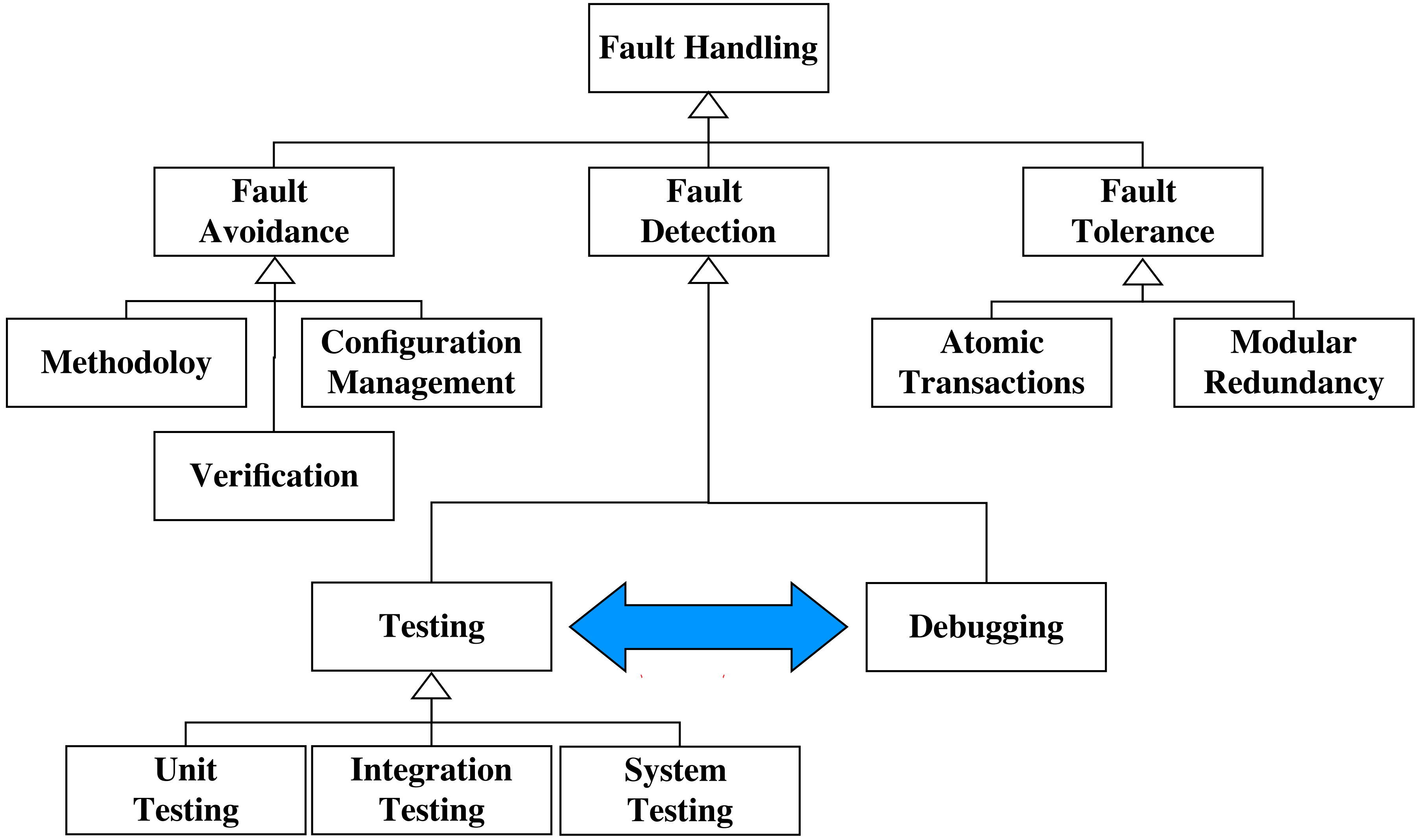
Testing



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

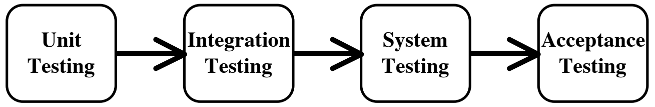
- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra).
- Testing is not for free

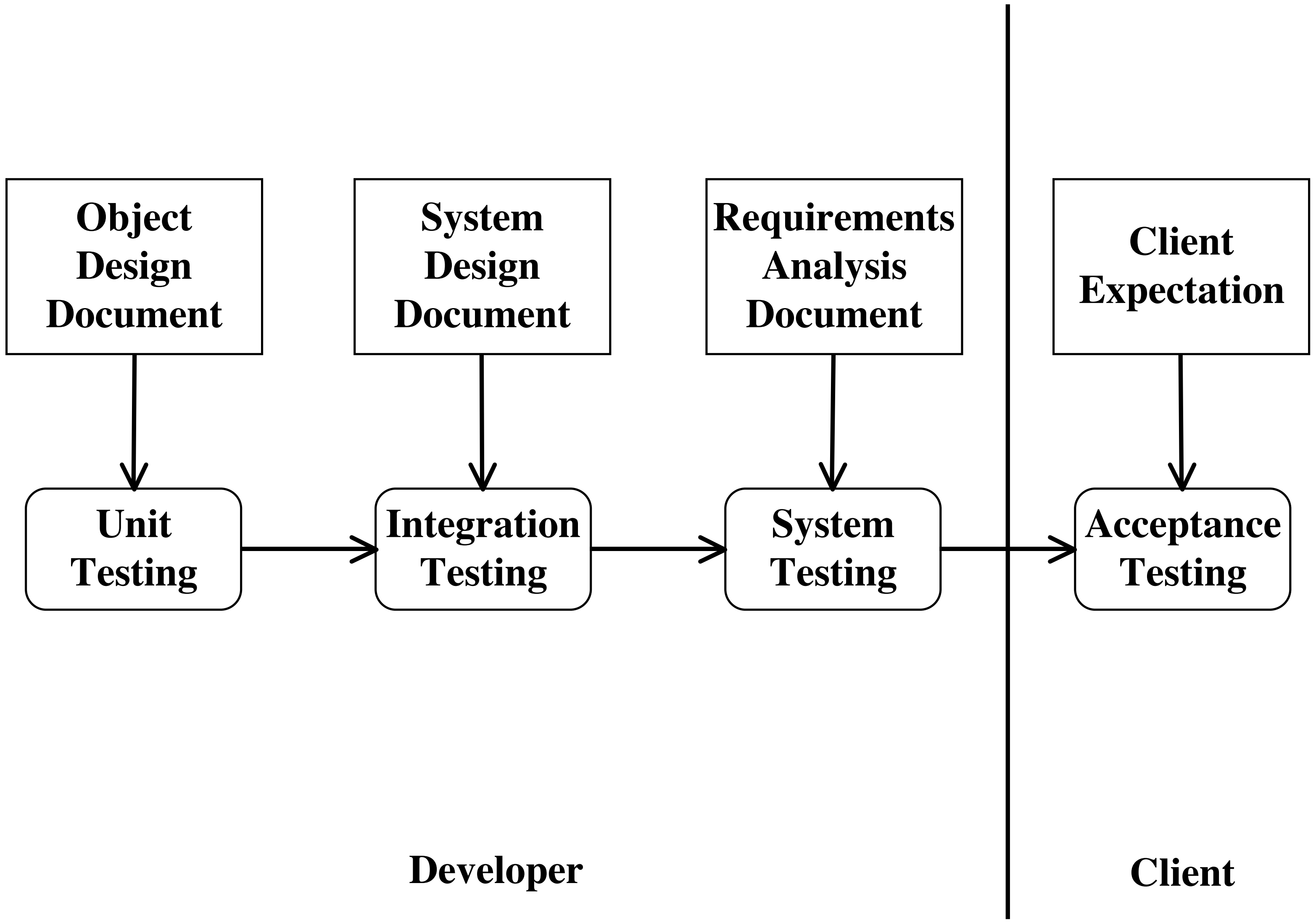
=> Define your goals and priorities

Testing takes creativity

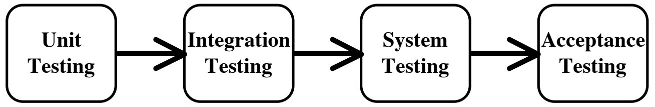
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent testers
  - We often develop a certain mental attitude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



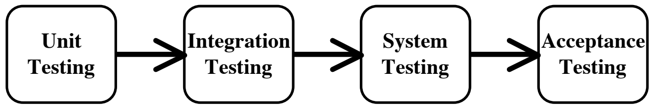


Types of Testing



- 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.

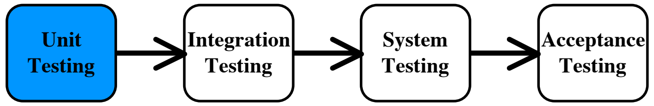
When should you write a test?

- Traditionally after the source code to be tested
- In XP before the source code to be tested
  - 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.

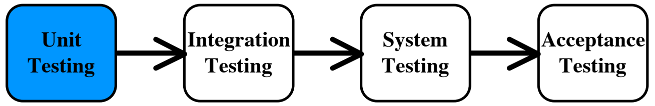
Unit Testing



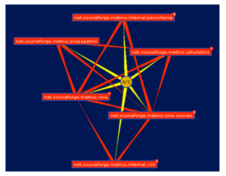
- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)

- Dynamic Testing (at run time)
  - Black-box testing
  - White-box testing.

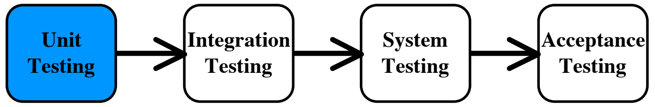
Static Analysis with Eclipse



- 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

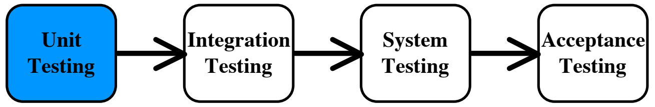
- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

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.

Status: Where are we now?



- Terminology
- Testing Activities
- Unit testing
    - Static Testing
    - Dynamic Testing
        - Blackbox Testing
            - Example...

Black box testing: An example

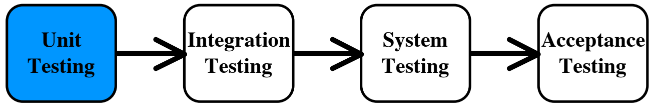
```
public class MyCalendar {

  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



- Test cases are derived from the internal structure (e.g. source code) of the tested unit

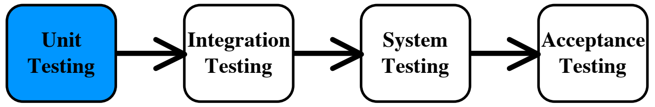
- Coverage Metrics:
  - Code coverage
  - Branch coverage
  - Condition coverage
  - Path coverage

=> Details in the exercise session about testing

Unit Testing Heuristics

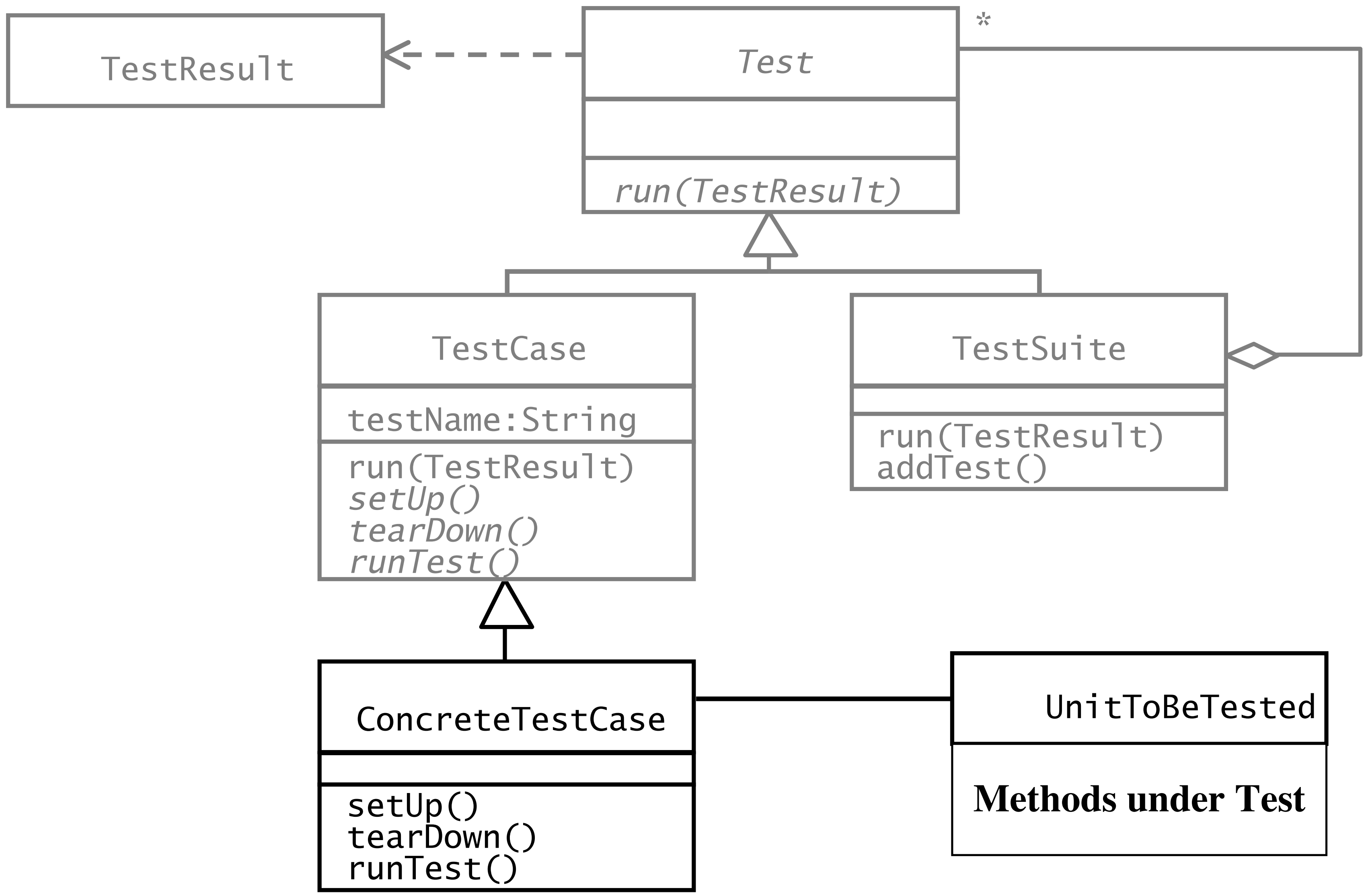
1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
2. Develop the test cases
   - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. 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



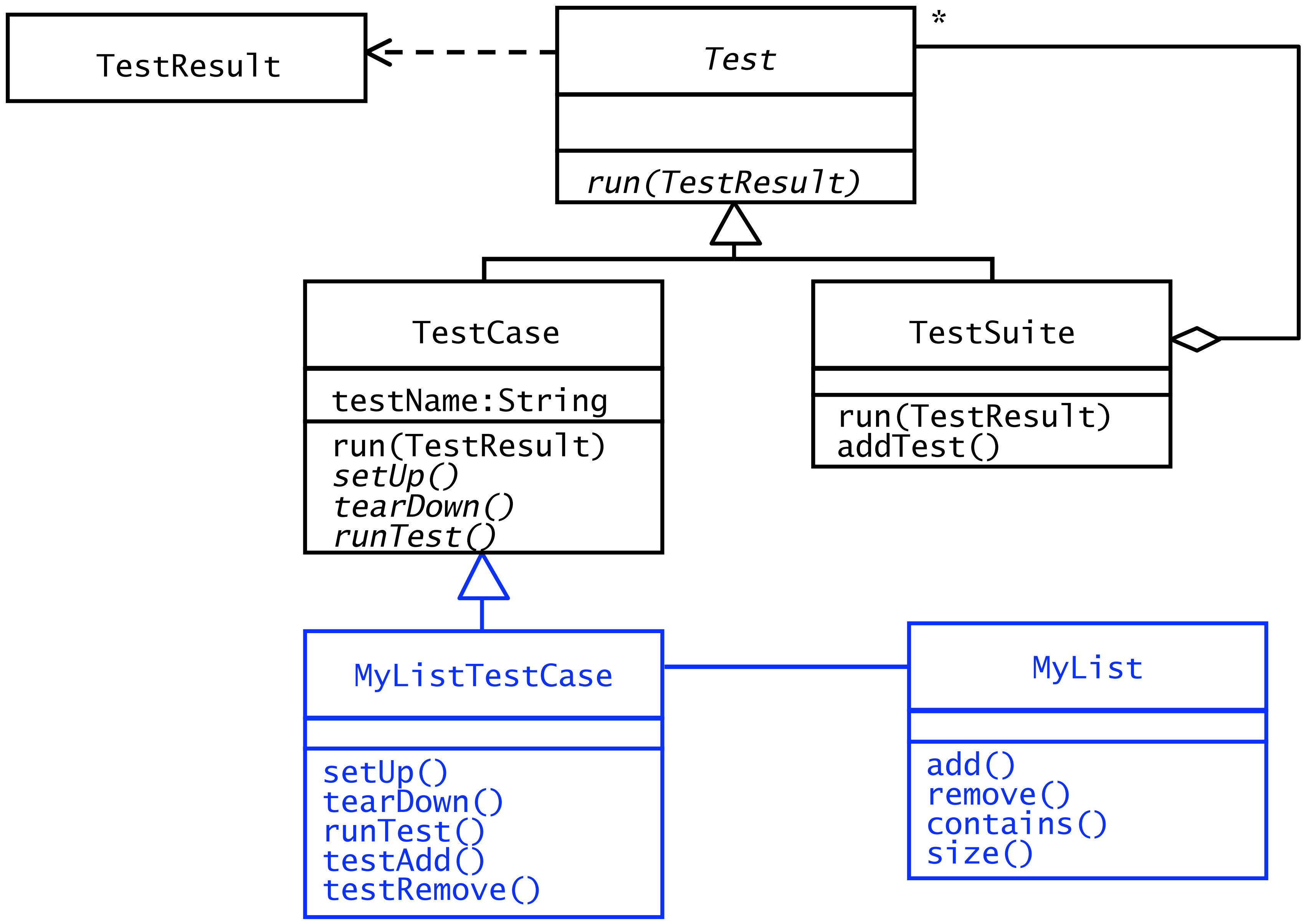
- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Written by Kent Beck and Erich Gamma
- Written with "test first" and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring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

JUnit Classes



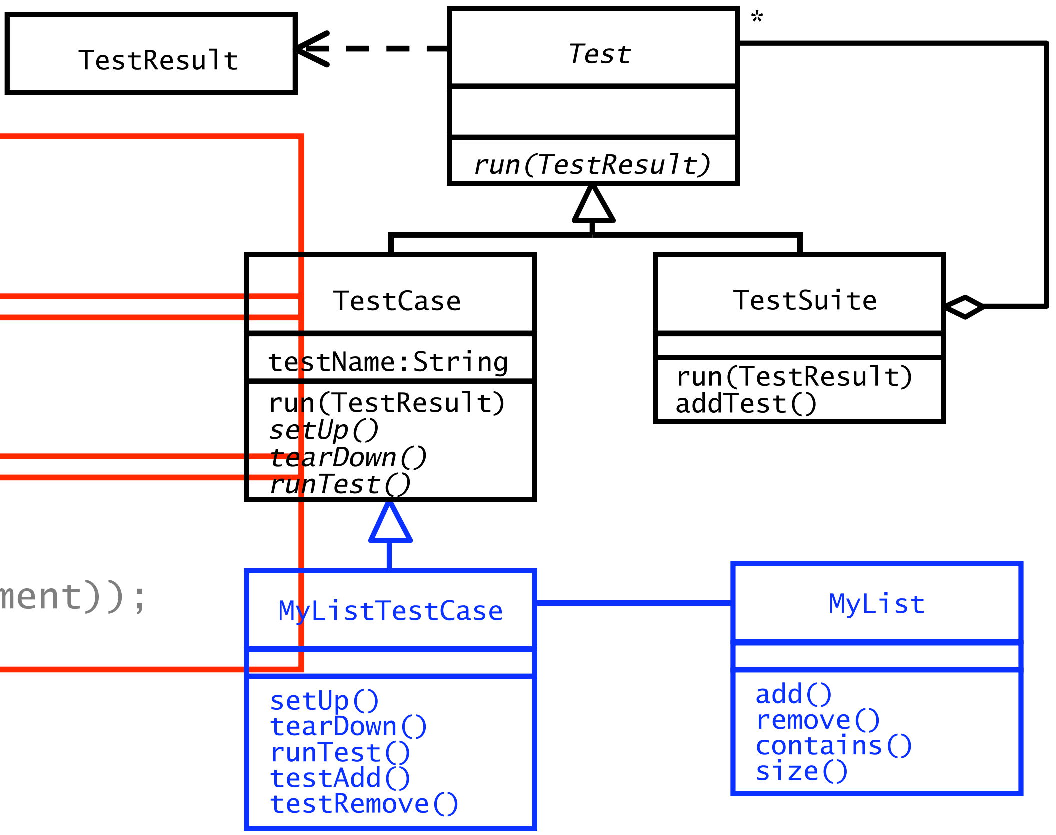
An example: Testing MyList

- 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”;
}
```

Test Fixture

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

Test Case

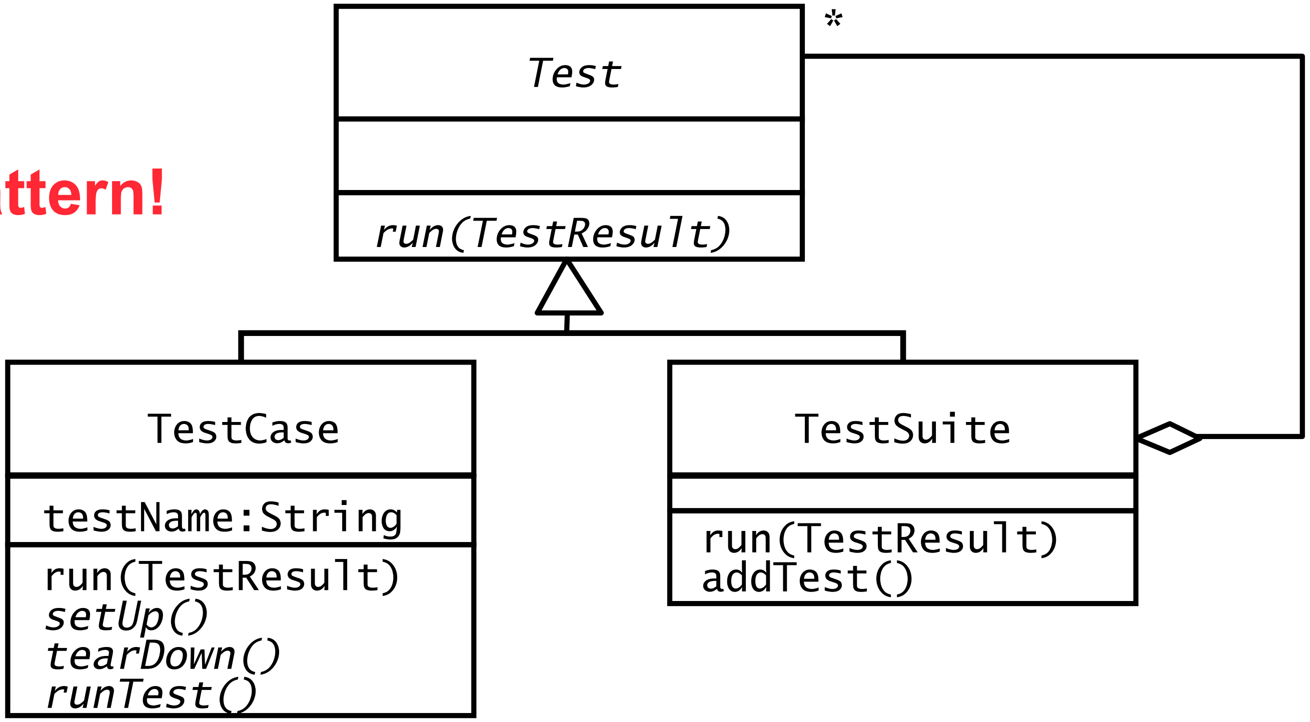
```
public void testRemove() {
   aList.add(anElement);
   aList.remove(anElement);
   assertTrue(aList.size() == 0);
   assertFalse(aList.contains(anElement));
}
```

Test Case

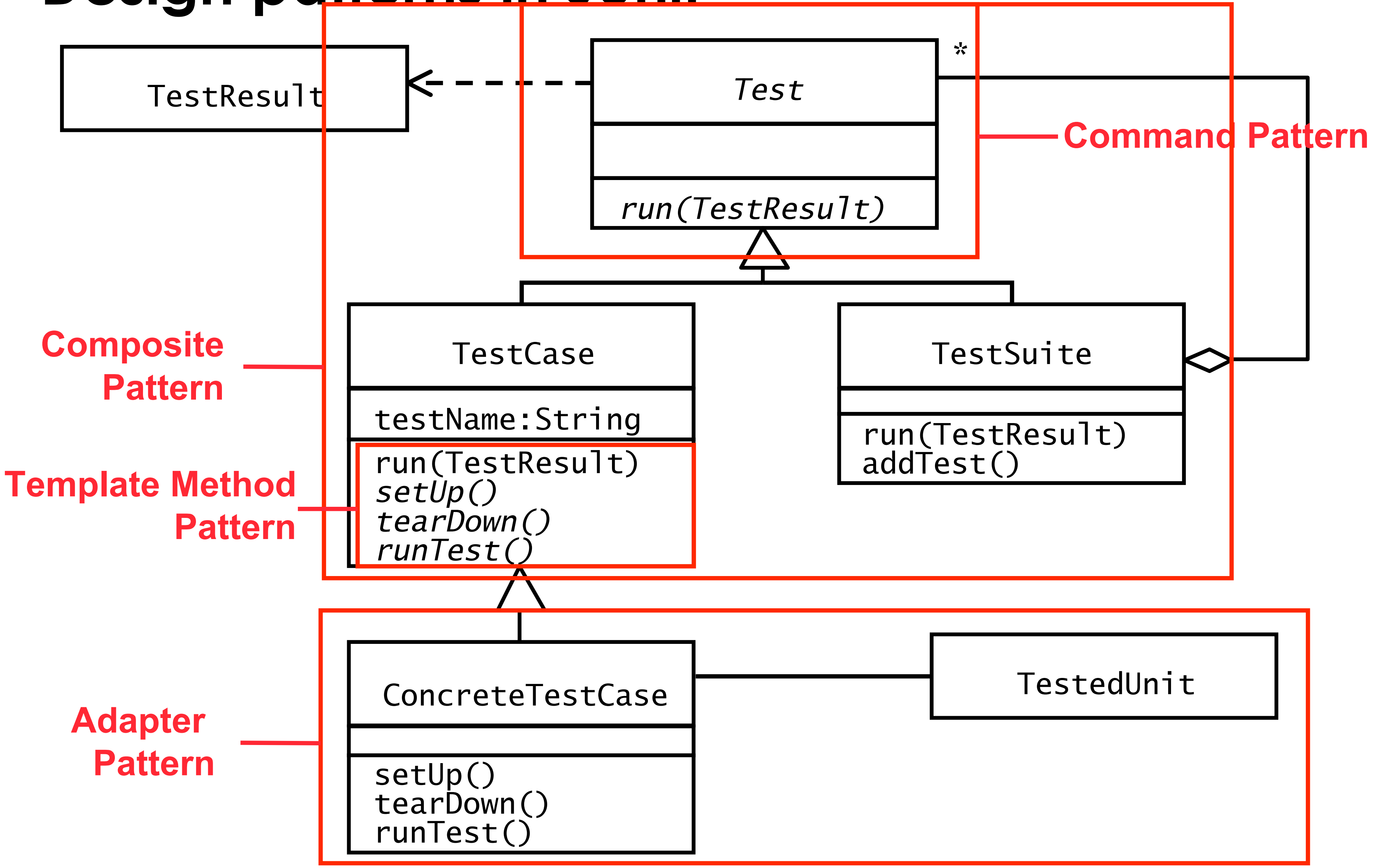
Collecting TestCases into TestSuites

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

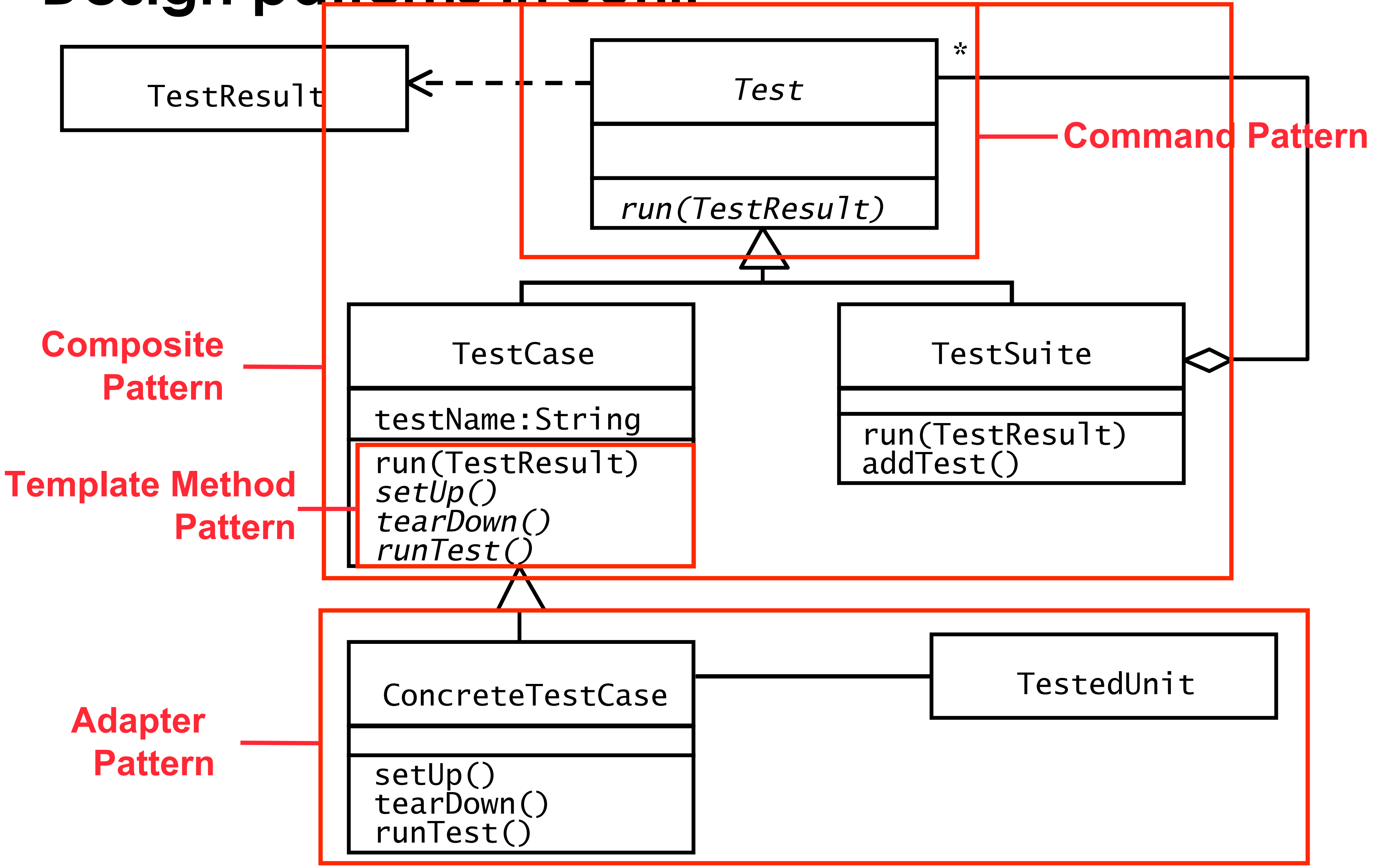
Composite Pattern!



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.