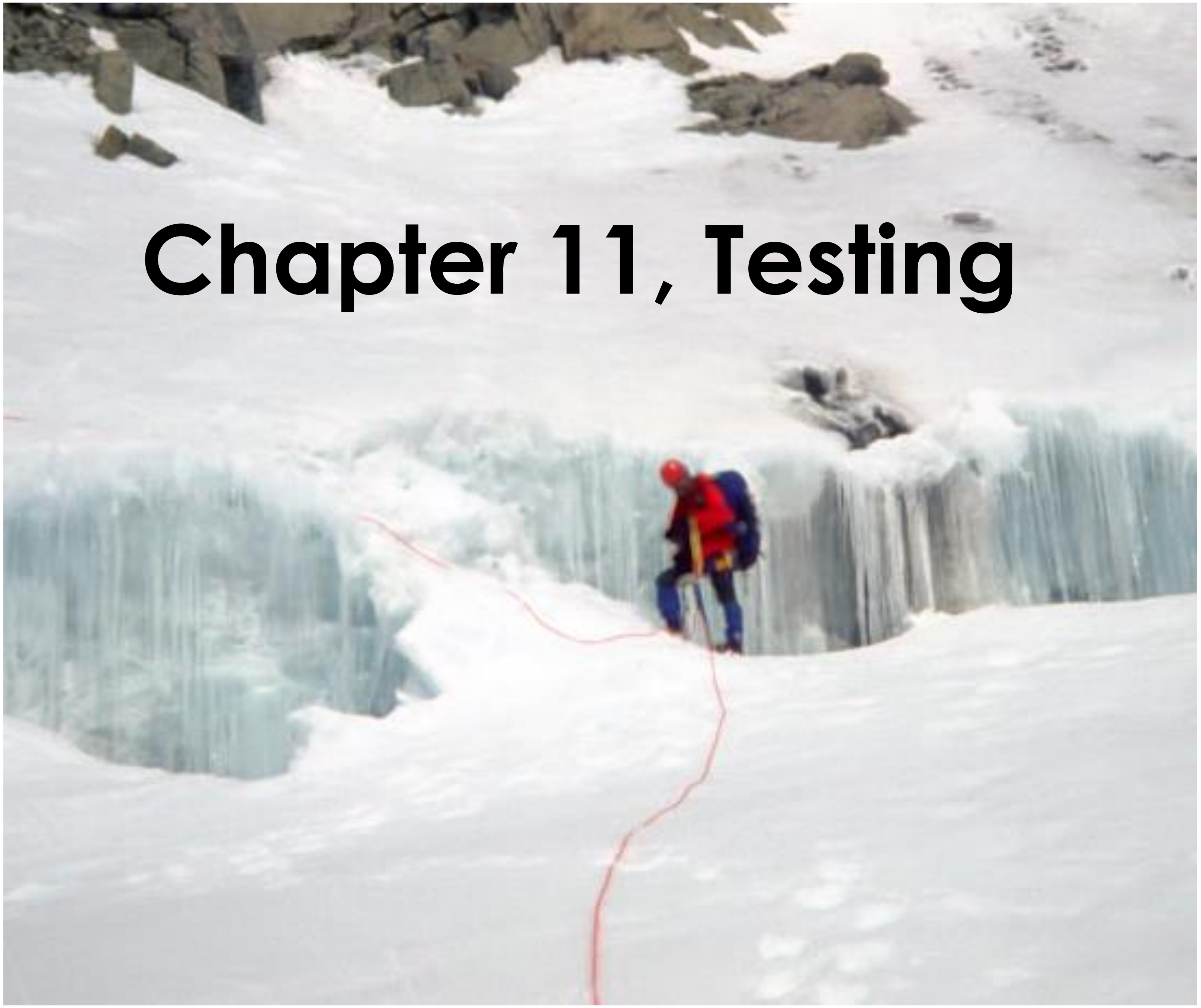
Object-Oriented Software Engineering
Using UML, Patterns, and Java



Chapter 11, Testing

Testing – the problem

Testing is 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**.

Testing – the problem_2

- **Performance testing** finds **differences between nonfunctional requirements** and **actual system performance**.
- When differences are found, developers identify the defect causing the observed failure and modify the system to correct it. In other cases, the system model is identified as the cause of the difference, and the system model is updated to reflect the requirements.
- **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 which are constructive activities.

Testing – the problem_3

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**.

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.

Testing – perception

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.
- 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.

Testing – overview

Reliability is a **measure of success** with which **the observed behavior** of a system **conforms to the specification of its behavior**.

- **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** is any **deviation of the observed behavior from the specified behavior**.

- An **erroneous state/error** means the **system is in a state** such that **further processing by the system will lead to a failure**, which then causes the system to deviate from its intended behavior.

Testing – overview_2

- A **fault/defect/bug**, is the **mechanical or algorithmic cause of an erroneous state**.
- 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."**

Testing activities

Test planning allocates resources and schedules the testing.

This activity should occur early in the development phase so that sufficient time and skill is dedicated to testing. Developers can design test cases as soon as the models they validate become stable.

- **Usability testing** tries to **find faults in the user interface design** of the system. Often, systems fail to accomplish their intended purpose simply because their users are confused by the user interface and unwillingly introduce erroneous data.

- **Unit testing** tries to **find faults in participating objects and/or subsystems** with respect to the use cases from the use case model.

Testing activities_2

- **Integration testing** is the activity of finding faults by testing individual components in combination.
- **Structural testing** is the culmination of integration testing involving all components of the system.
- **Integration tests** and **structural tests** exploit knowledge from the **SDD (System Design Document)** using an integration strategy described in the **Test Plan (TP)**.
- **System testing** tests all the components together, seen as a single system to identify faults with respect to the scenarios from the problem statement and the requirements and design goals identified in the analysis and system design, respectively:

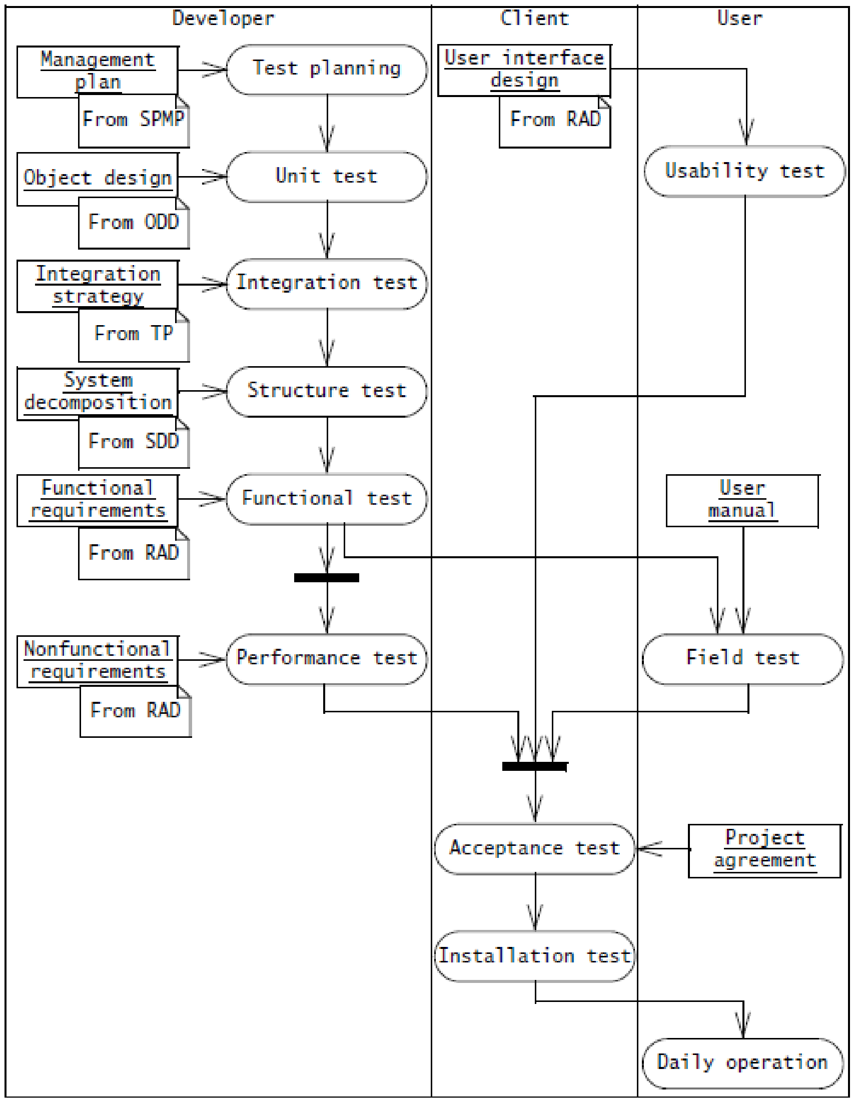
Testing activities_3

- **Functional testing** tests the requirements from the RAD and the user manual.

- **Performance testing** checks the **nonfunctional requirements** and **additional design goals from the SDD**. Functional and performance testing are done by developers.

- **Acceptance testing** and **installation testing** check the system against the project agreement and is done by the client, if necessary, with help by the developers.

Testing activities and assoc. stakeholders



SPMP - Software Project Management Plan
TP - Test Plan
SDD - System Design Document
RAD - Requirements Analysis Document

Testing concepts

- **Test component** is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.

- A **fault/bug/defect**, is a **design or coding mistake** that may cause **abnormal component behavior**.

- An **erroneous state** is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.

- A **failure** is a **deviation between the specification and the actual behavior**. A failure is triggered by one or more erroneous states. Not all erroneous states trigger a failure.

Testing concepts_2

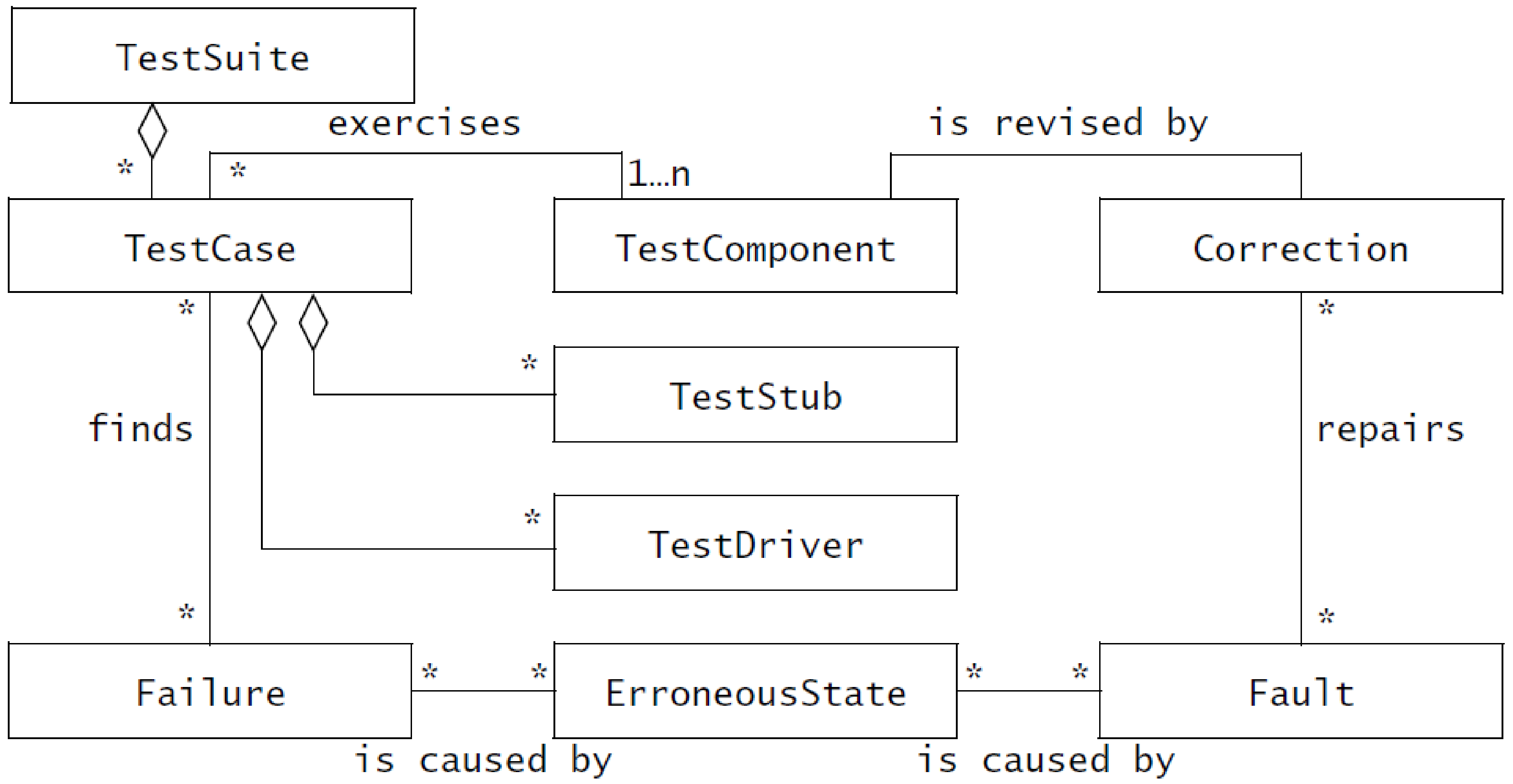
- A **test case** is a **set of inputs and expected results** that exercises a test component with the purpose of causing failures and detecting faults.

- A **test stub** is a **partial implementation of components on which the tested component depends**.

- A **test driver** is a **partial implementation of a component that depends on the test component**. **Test stubs** and **drivers** enable components to be isolated from the rest of the system for testing.

- A **correction** is a **change to a component**. The **purpose of a correction is to repair a f**ault. Note that a correction can introduce new faults.

Model elements used during testing



Attributes of the class TestCase

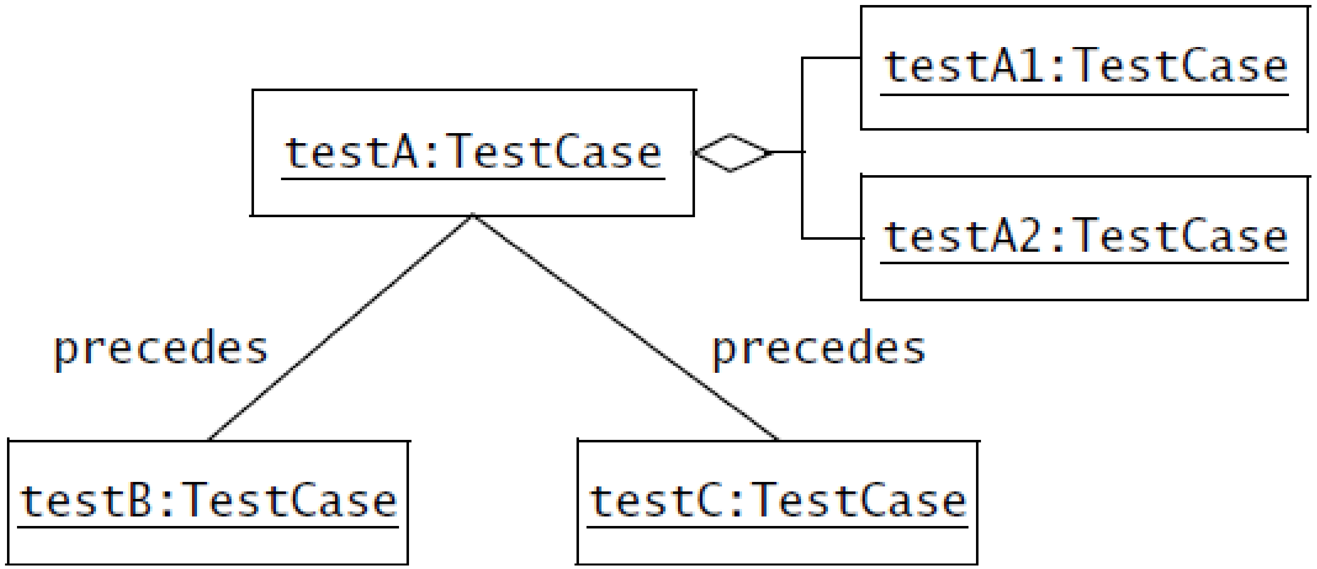
A **test case** is a **set of input data and expected results** that exercises a component with the purpose of causing failures and detecting faults. A test case has **five attributes**: **name**, **location**, **input**, **oracle**, and **log**

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

Test model width test cases

Once test cases are identified and described, relationships among test cases are identified.

Aggregation and the **precede associations** are used to describe the relationships between the test cases. Aggregation is used when a test case can be decomposed into a set of subtests. Two test cases are related via the precede association when one test case must precede another test case.



Test model width test cases

Test cases are classified into **blackbox tests** and **whitebox tests**:

Blackbox tests focus on the **input/output behavior** of the component. **Blackbox tests do not deal with the internal aspects of the component**, nor with the behavior or the structure of the components.

Whitebox tests focus on the internal structure of the component. A **whitebox test** makes sure that, independently from the particular input/output behavior, **every state in the dynamic model of the object and every interaction among the objects is tested**.

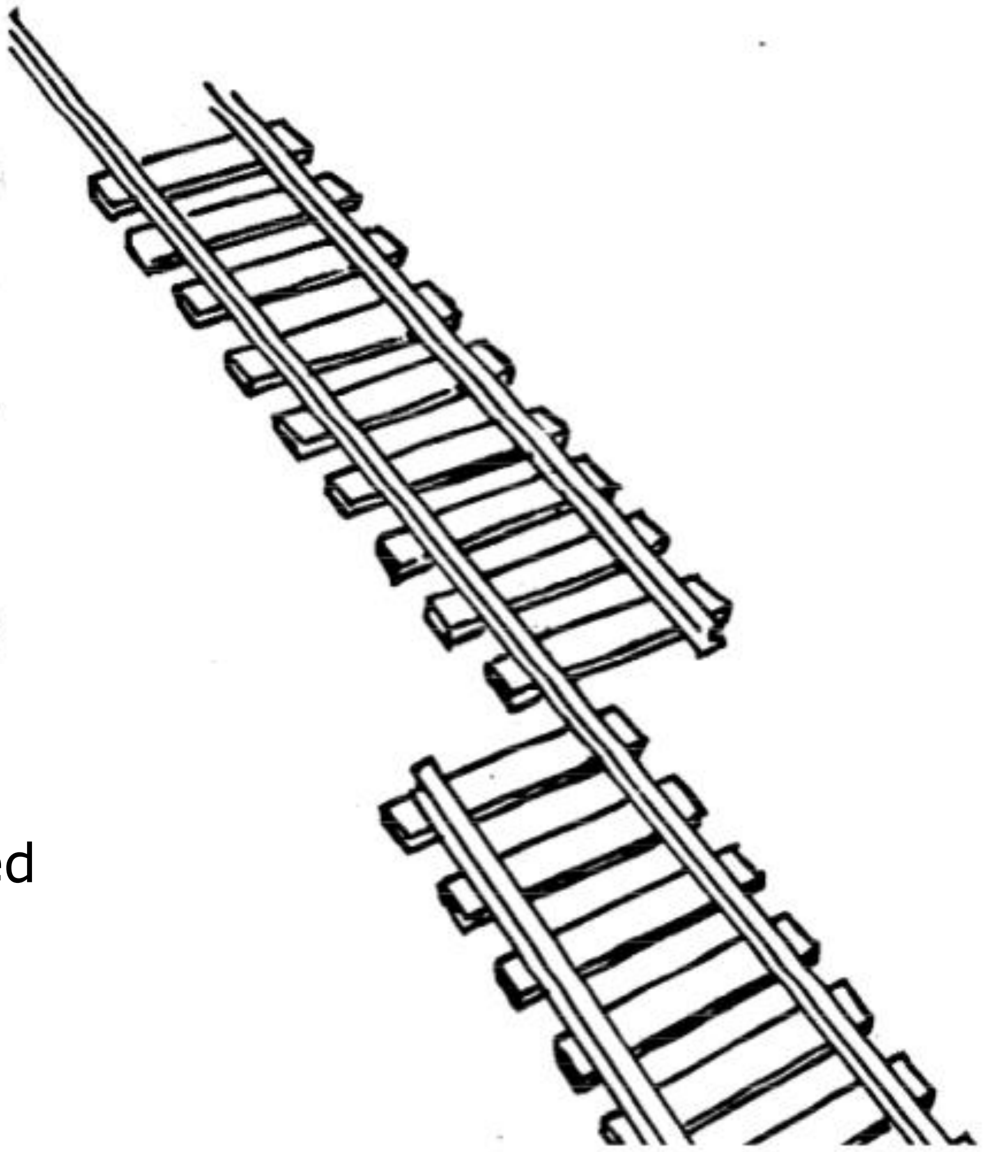
Test model width test cases_2

As a result, **whitebox testing goes beyond blackbox testing**. In fact, most of the whitebox tests require input data that could not be derived from a description of the functional requirements alone.

Unit testing combines both testing techniques: **blackbox testing** to test the functionality of the component, and **whitebox testing** to test structural and dynamic aspects of the component.

Executing test cases on single components or combinations of components **requires** the **tested component to be isolated from the rest of the system**. **Test drivers** and **test stubs** are used to substitute for missing parts of the system.

What is this?



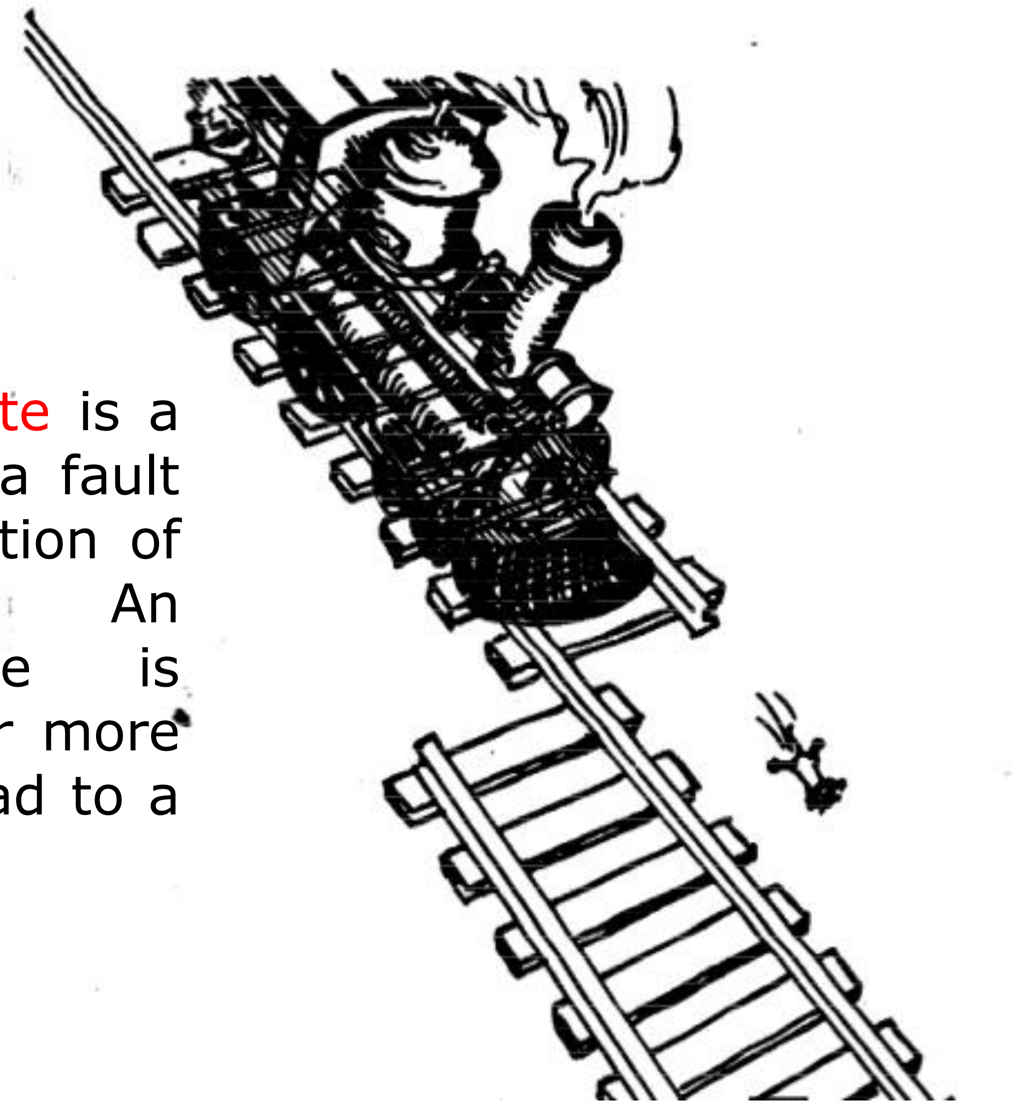
A failure?

An error?

A fault?

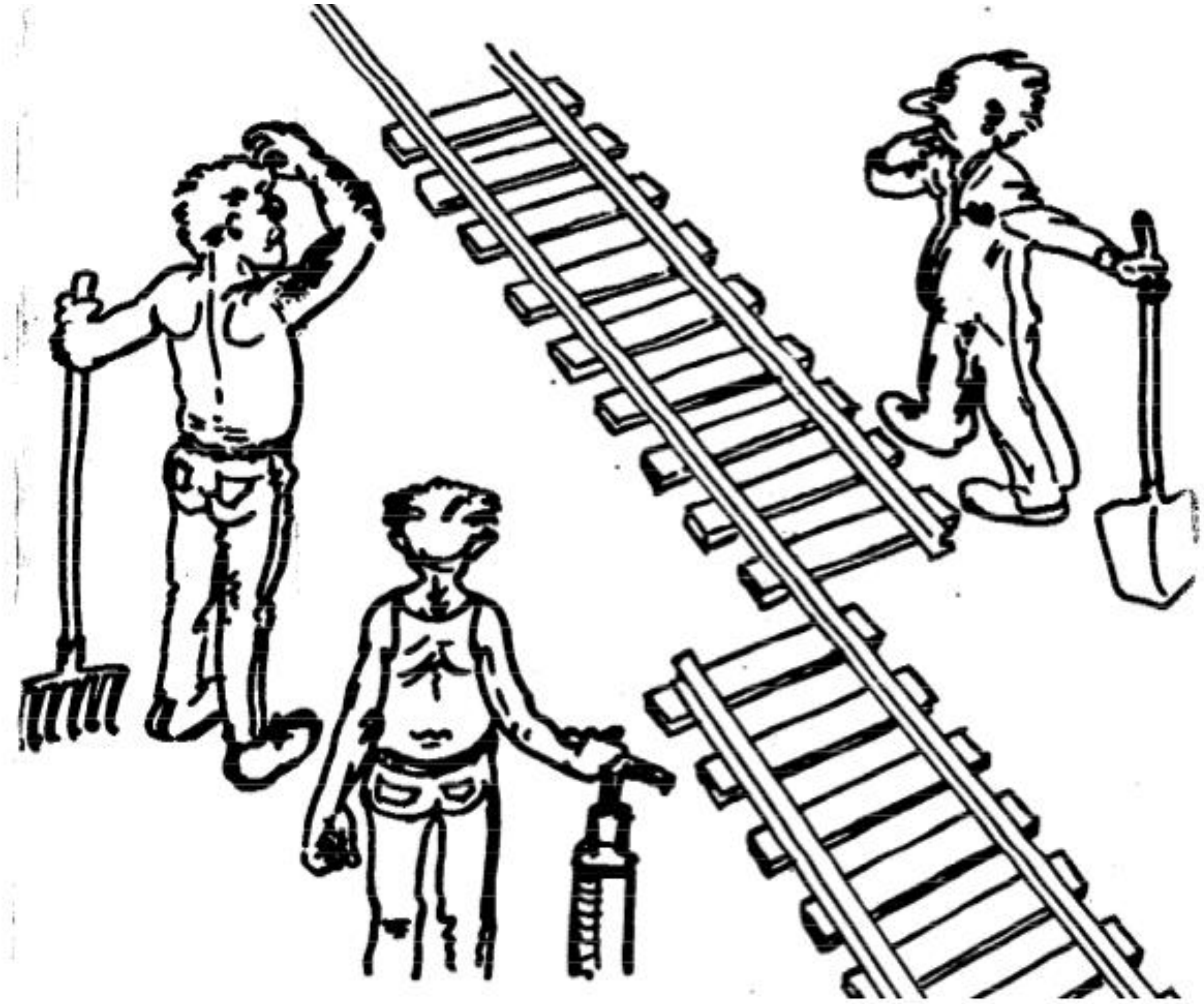
We need to describe specified and desired behavior first!

Erroneous State ("Error")



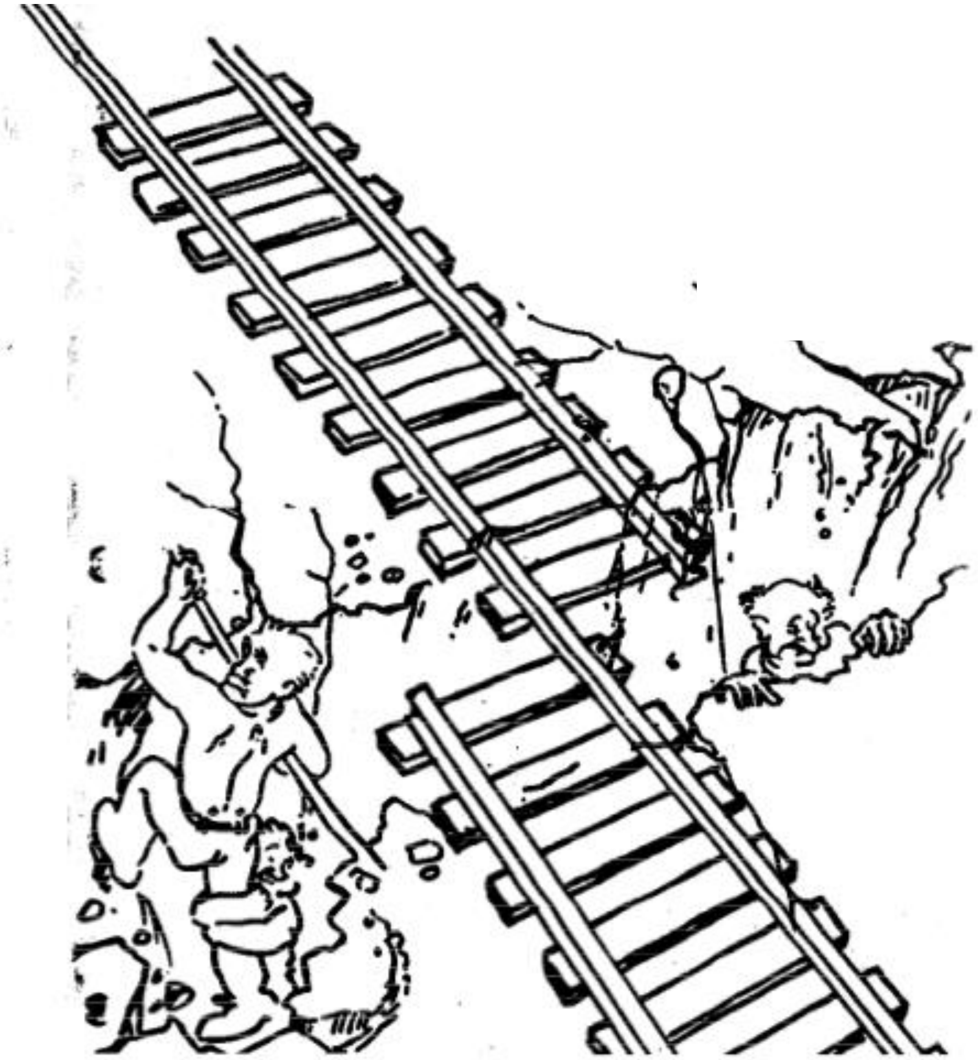
An erroneous state is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.

Algorithmic Fault



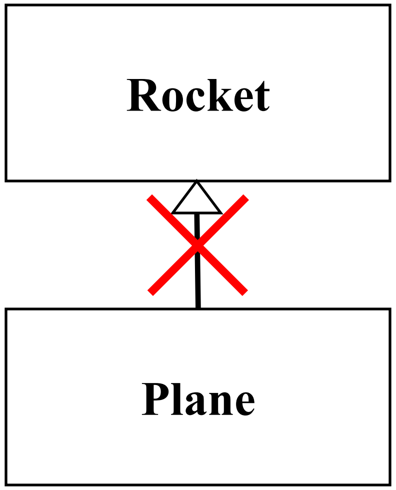
Mechanical Fault

A fault can have a mechanical cause, such as an earthquake



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.



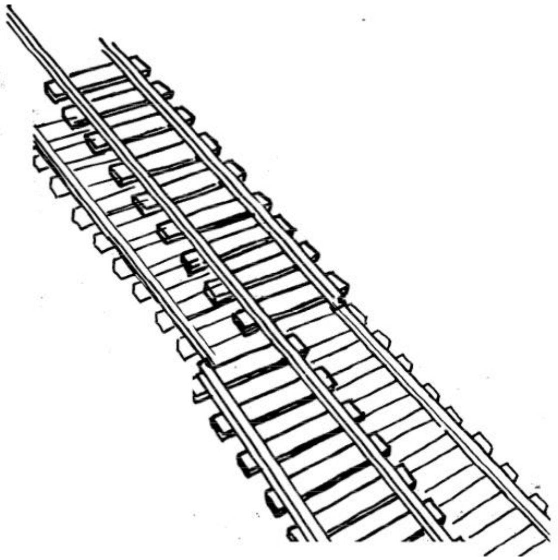
Examples of Faults and Errors

- 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.

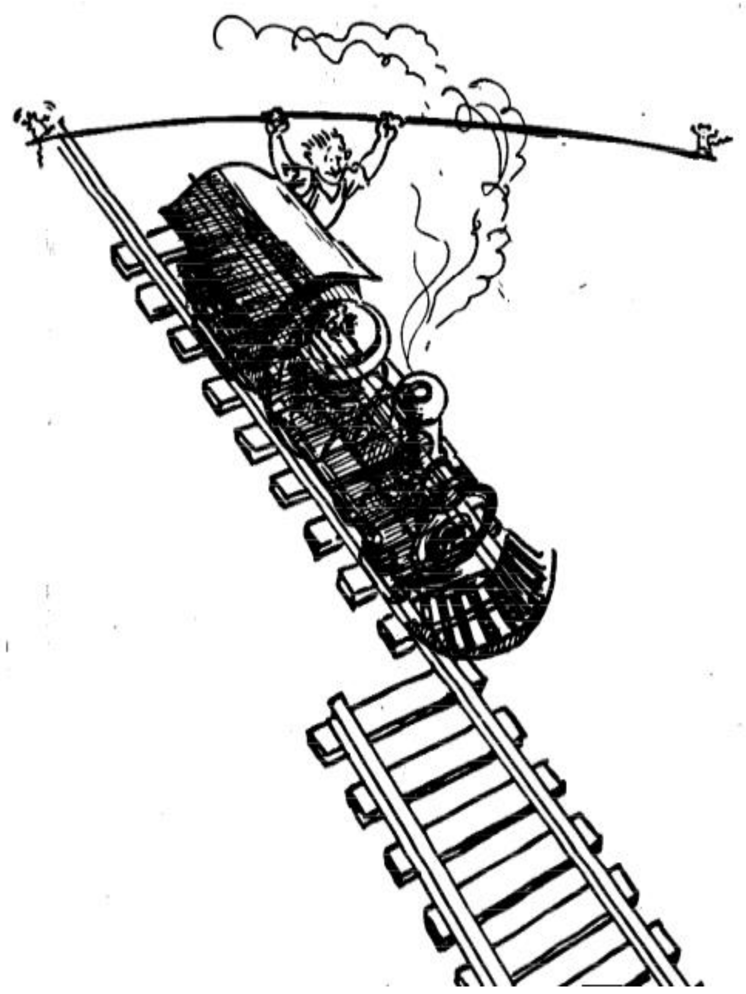
How do we deal with Errors, Failures and Faults?

Modular Redundancy



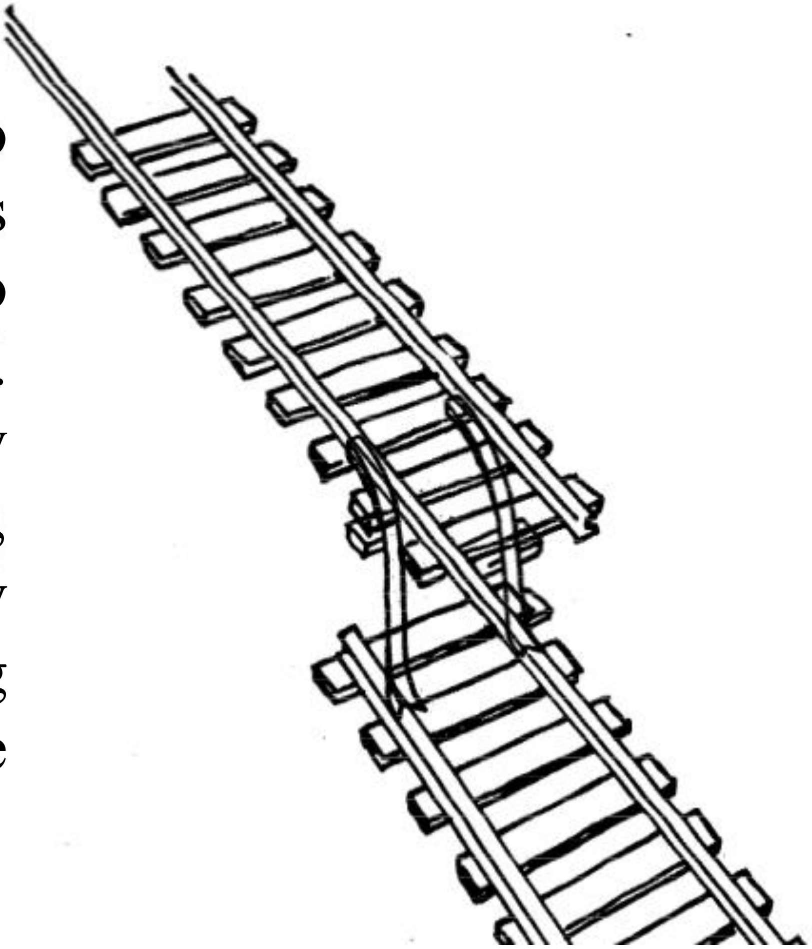
The Saturn rocket that launched the Apollo spacecraft used triple modular redundancy for the guidance system; that is, each of the components in the computer was tripled. The failure of a single component was detected when it produced a different output than the other two.

Declaring the Bug as a Feature

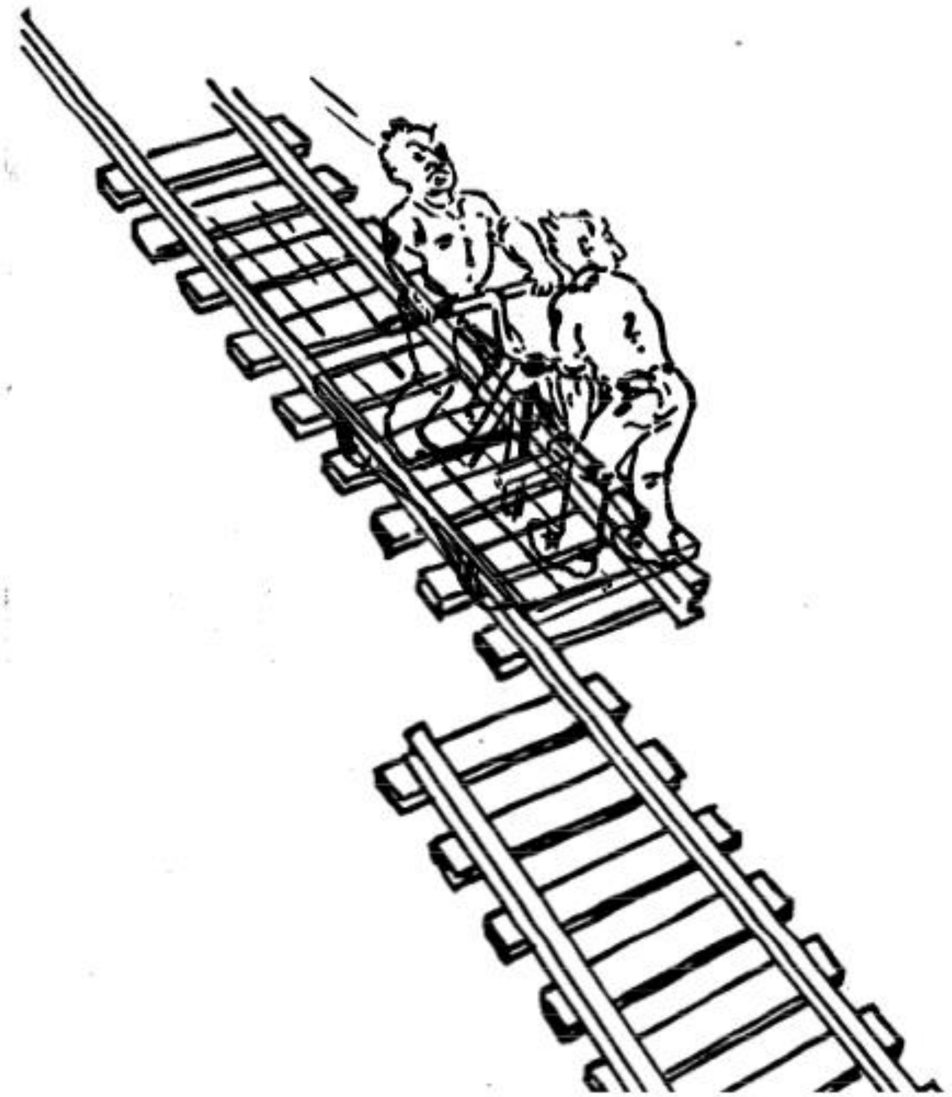


Patching

A **patch** is a set of changes to a **computer** program or its supporting data designed to update, fix, or improve it. This includes fixing security vulnerabilities and other bugs, with such **patches** usually being called bugfixes or bug fixes, and improve the usability or performance.



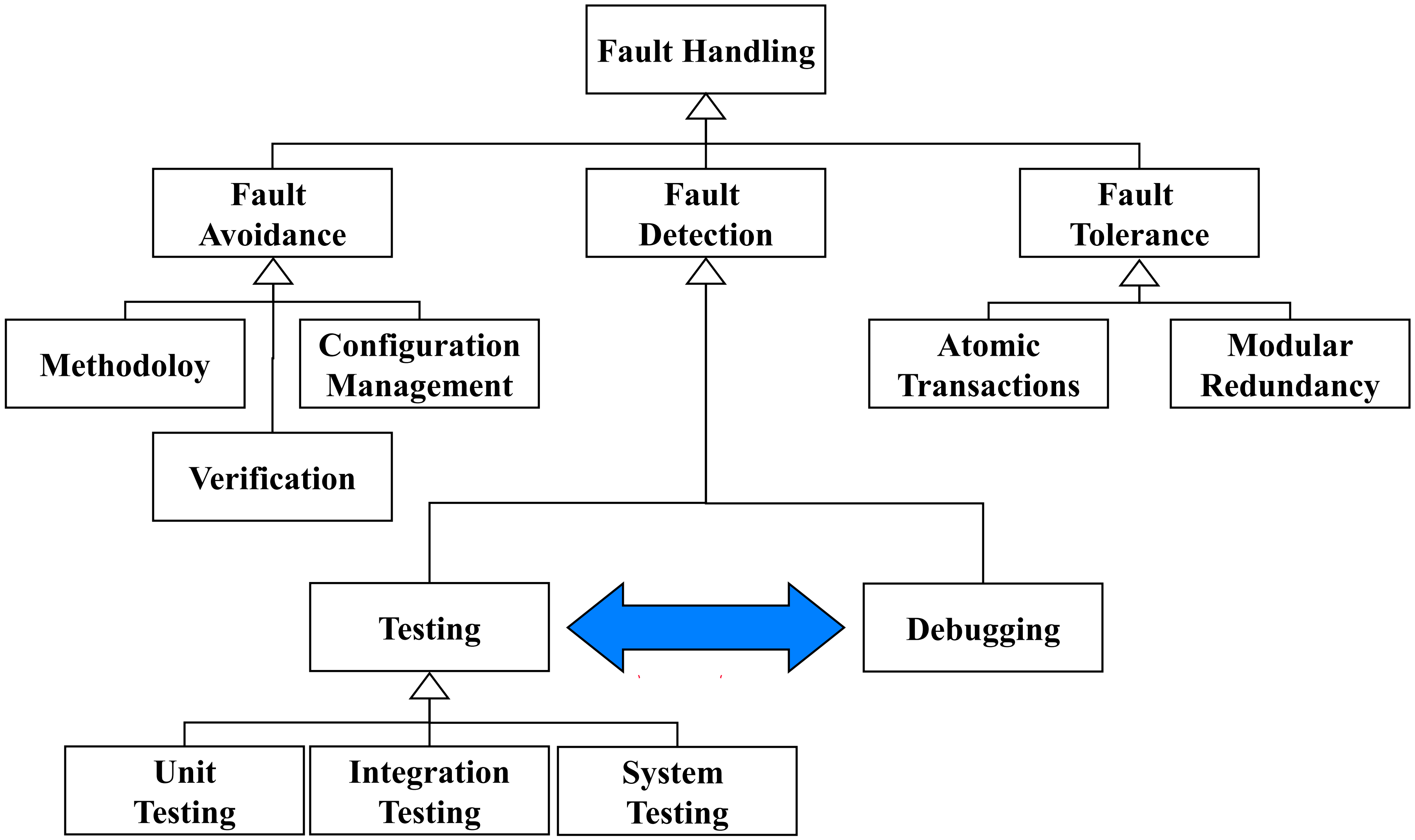
Testing



Another View on How to Deal with Faults

- Fault avoidance (before the system release)
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- Fault detection (while system is running)
  - 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 (recover from failure after release)
  - 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**

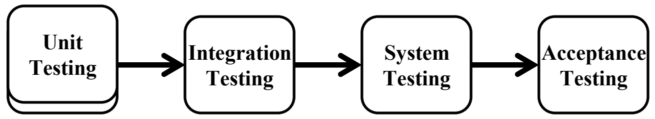
=> Define your goals and priorities

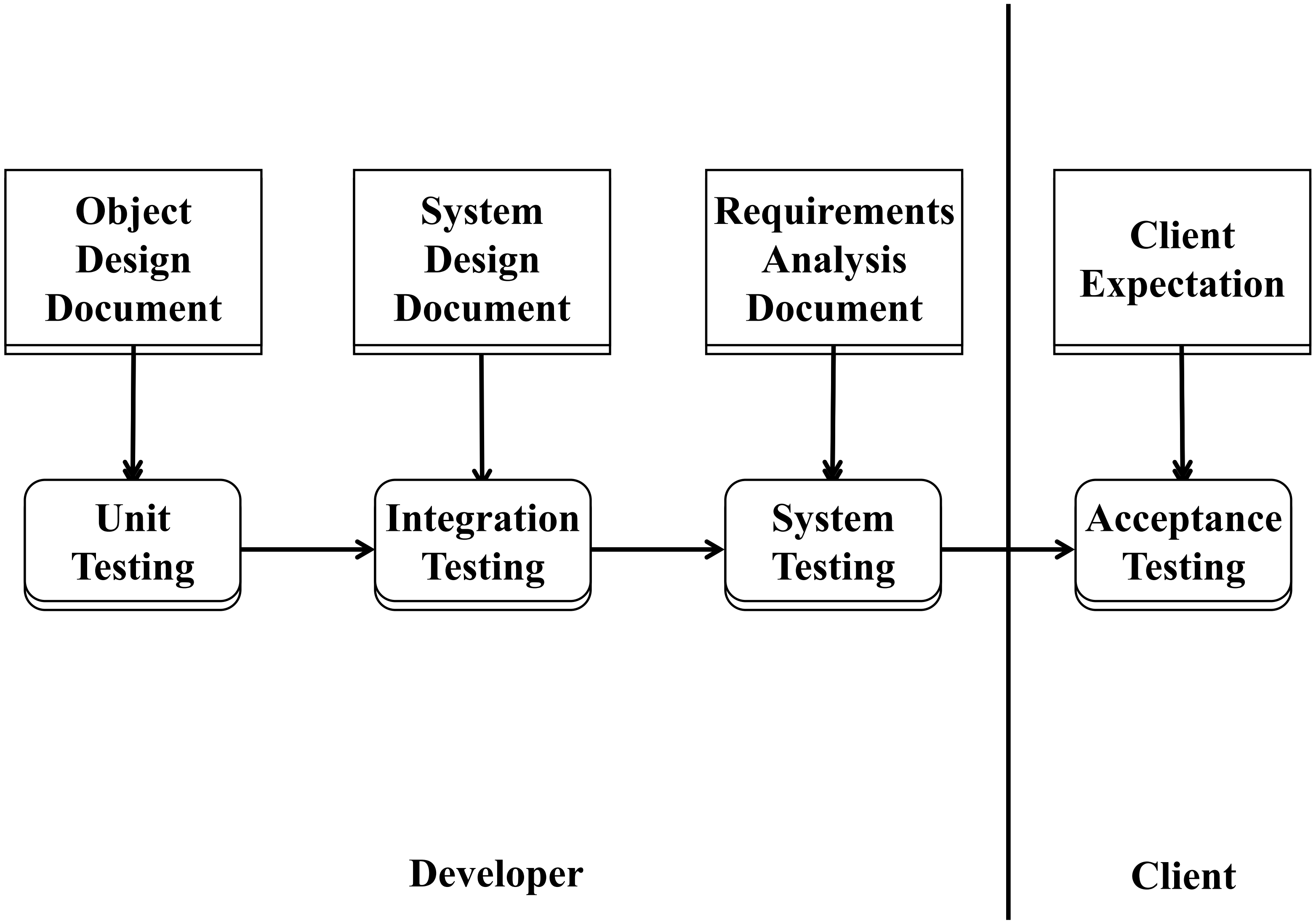
Testing takes creativity

- 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

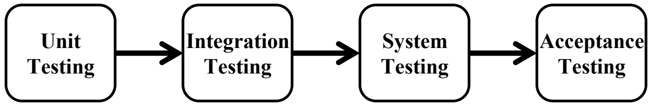
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should behave a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

Testing Activities



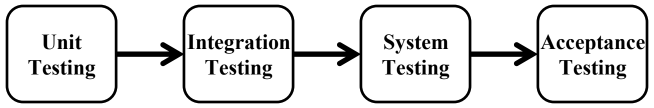


Types of Testing


$$\begin{tikzcd}\boxed{\text{Unit Testing}} \arrow[r] & \boxed{\text{Integration Testing}} \arrow[r] & \boxed{\text{System Testing}} \arrow[r] & \boxed{\text{Acceptance Testing}}\end{tikzcd}$$

- 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)

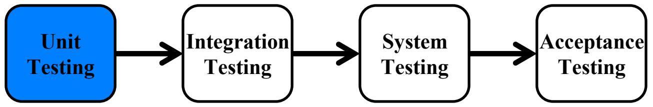
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

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.

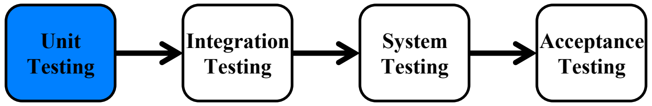
Unit Testing



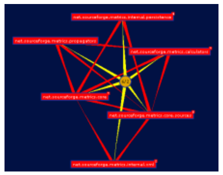
- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)

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

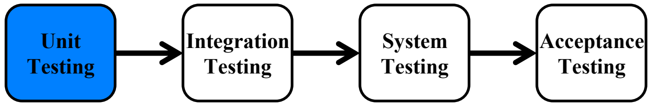
Static Analysis with Eclipse



- 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`:

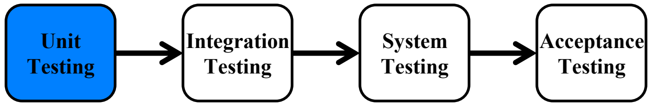
1: January, 2: February, …., 12: December

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



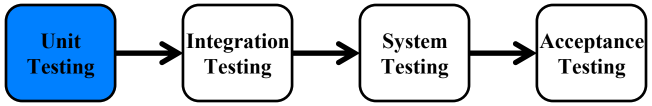
- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

=> 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. Double 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

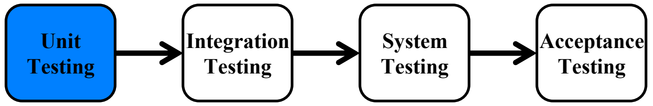


- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner

- A **test fixture** (also known as a **test** context) is the set of preconditions or state needed to run a **test**. The developer should set up a known good state before the **tests**, and return to the original state after the **tests**

- Written by Kent Beck and Erich Gamma

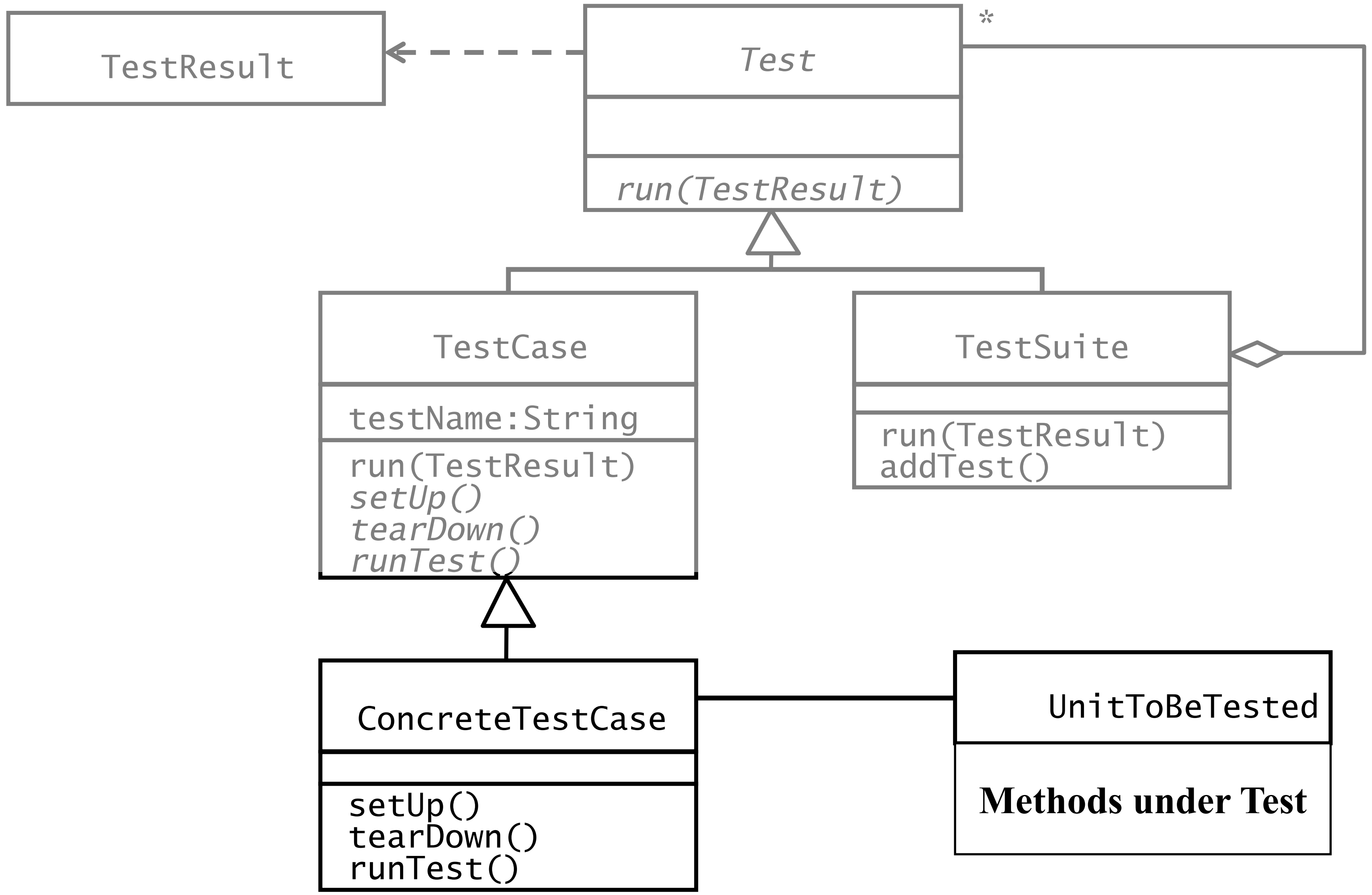
JUnit: Overview_2



- Written with “test first” and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring

- JUnit is Open Source
  - https://junit.org/junit5/
  - JUnit Version 5, was released on OCT 24, 2017

JUnit Classes

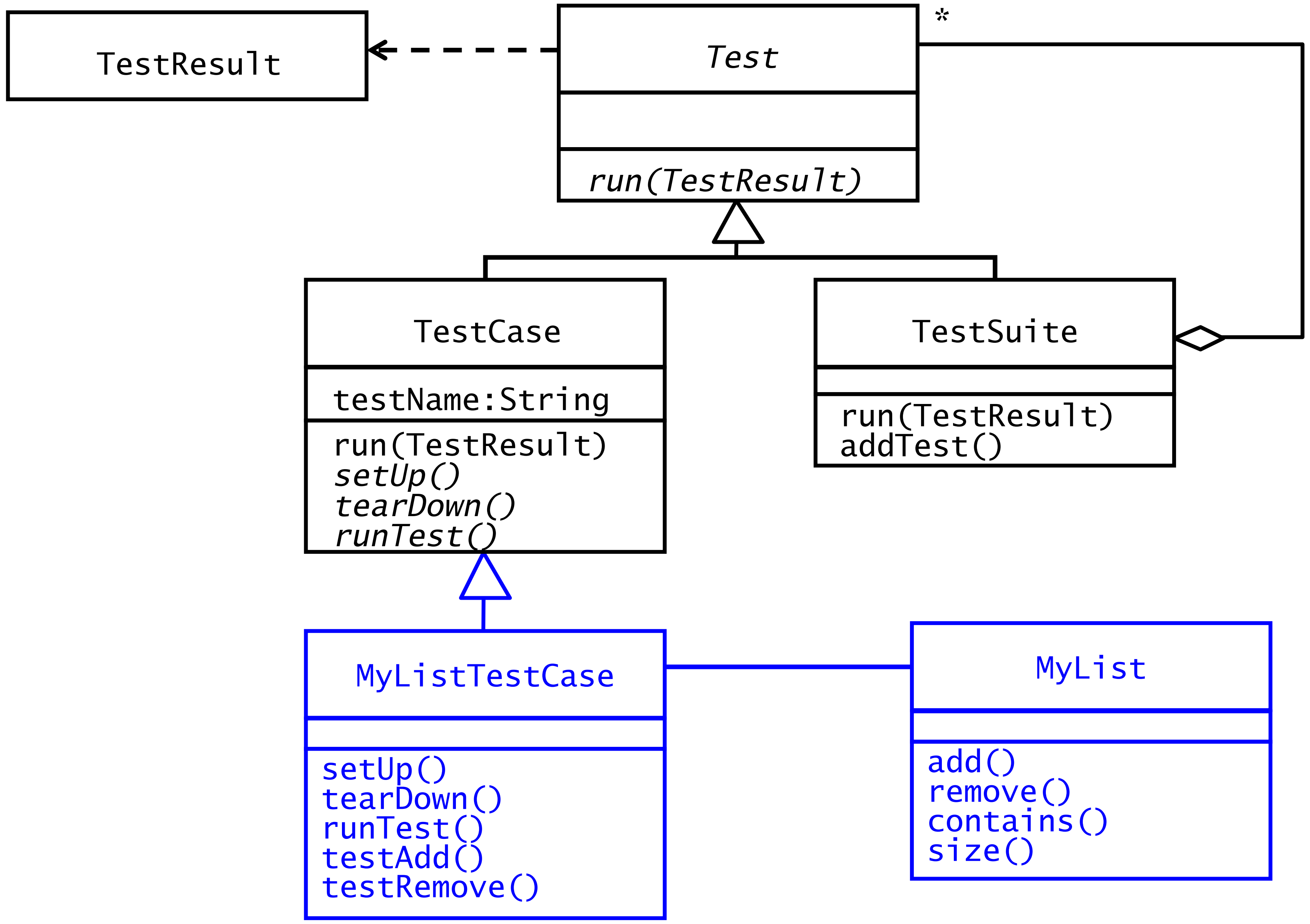


An example: Testing MyList

- 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()

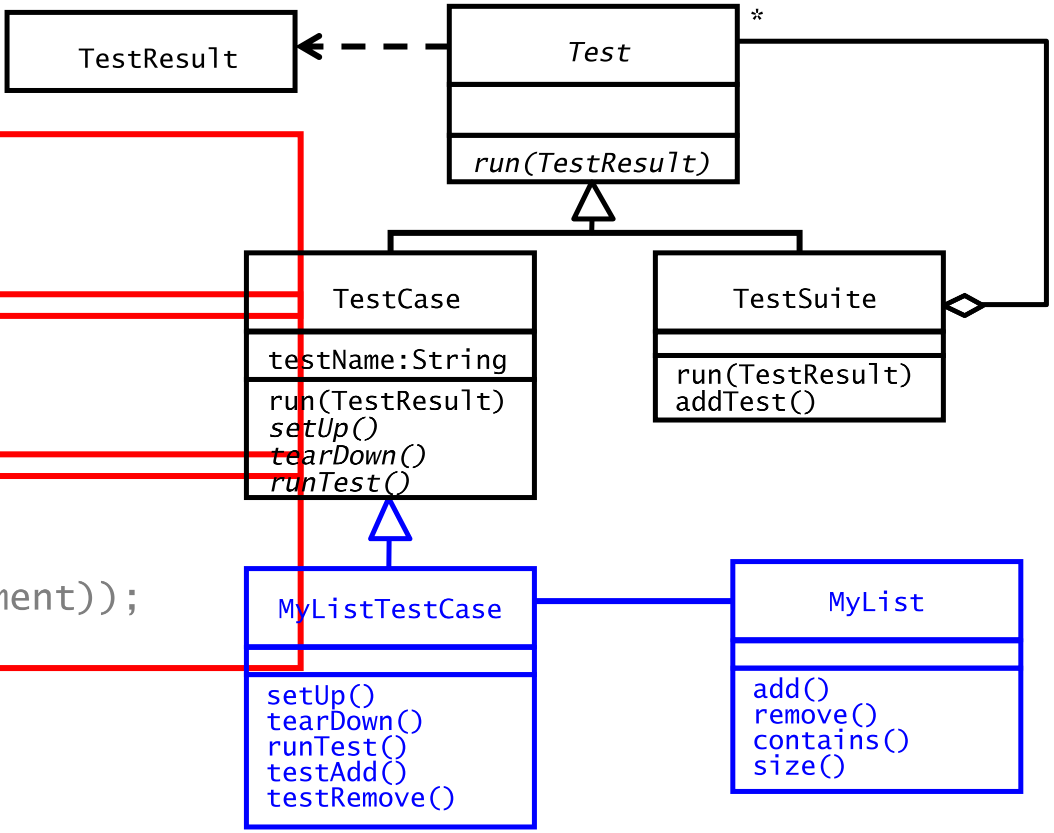
Writing TestCases in JUnit

```
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

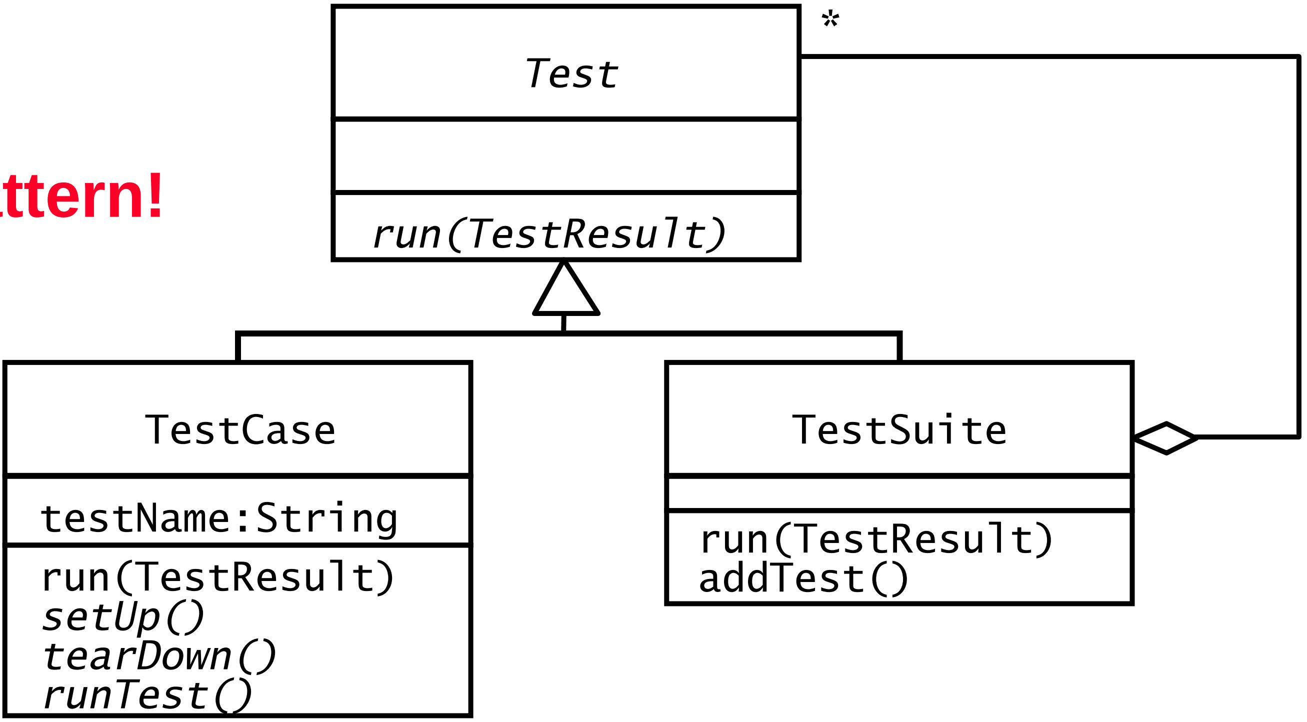
Test Case

Test Case

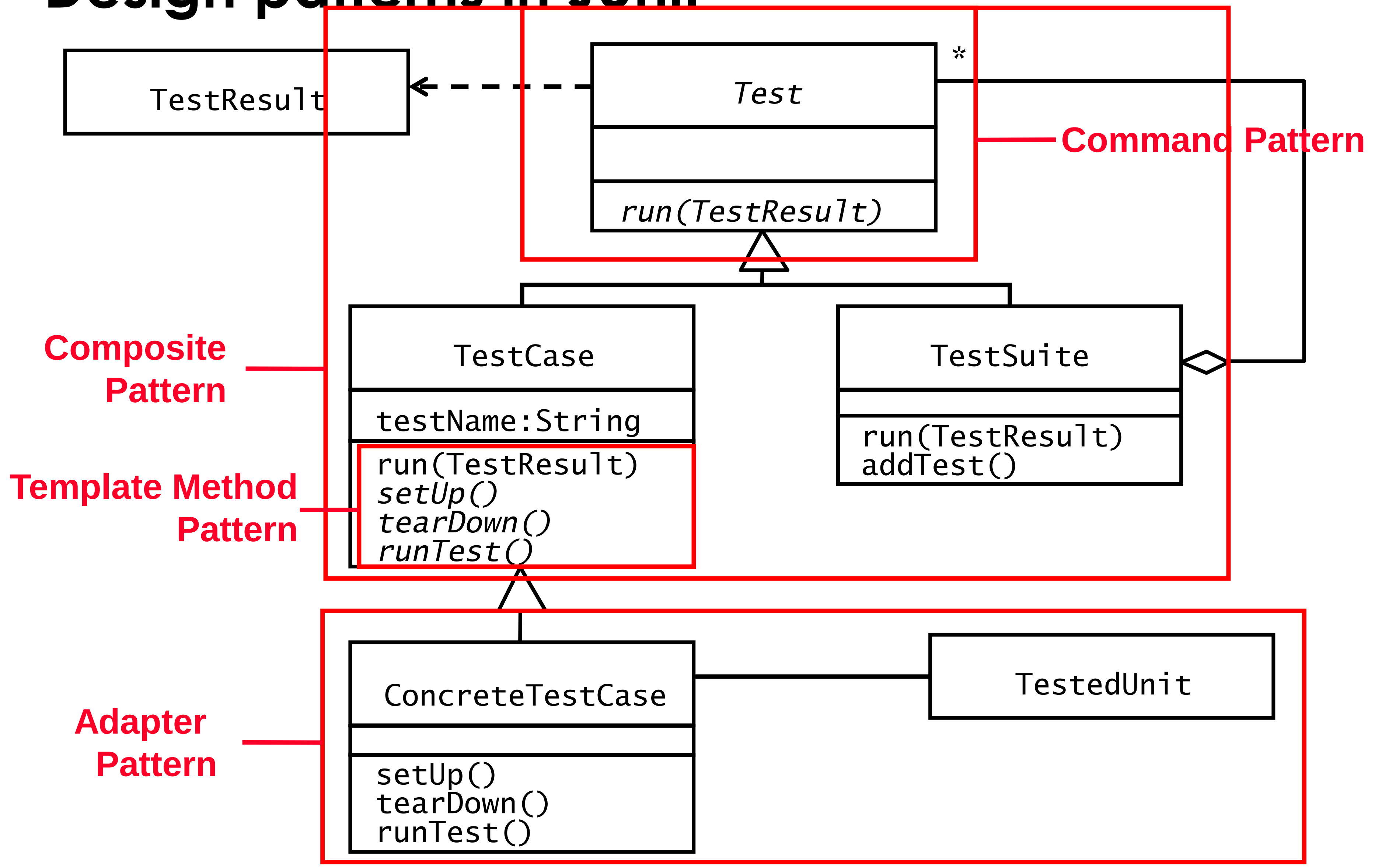
Collecting TestCases into TestSuites

```
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



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
  - 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 junit.org/junit5