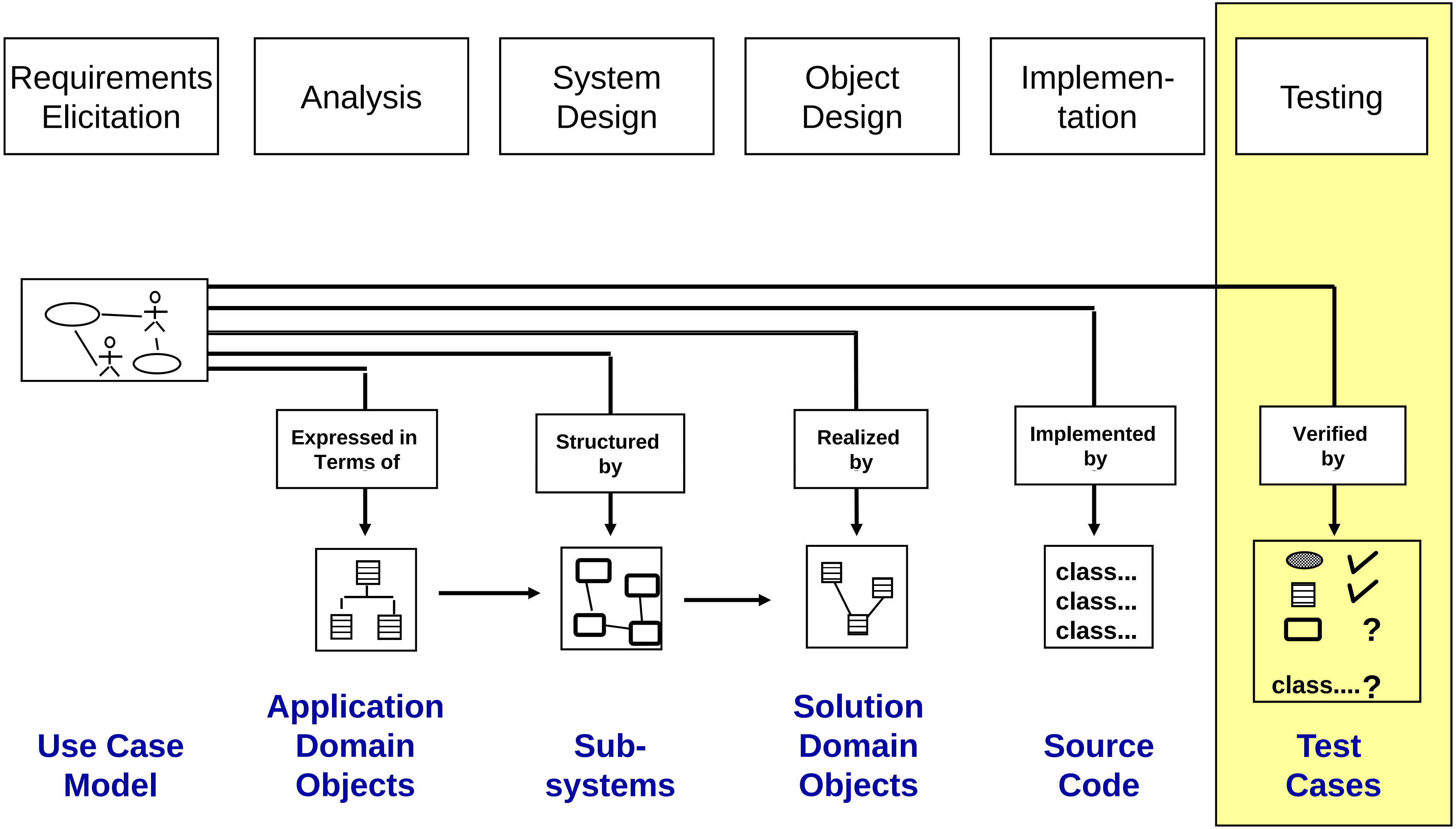
Chapter 13: Testing - 1

Object-**O**riented

Software **C**onstruction

Armin B. Cremers, Dr. Sascha Alda & Tobias Rho
(based on Bruegge & Dutoit)

Software Lifecycle Activities ...and their models



Requirements Elicitation

Analysis

System Design

Object Design

Implementation

Testing

Expressed in Terms of

Structured by

Realized by

Implemented by

Verified by

Use Case Model

Application Domain Objects

Sub-systems

Solution Domain Objects

Source Code

Test Cases

Testing

- Testing is the process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system
- Goal: Design tests that exercise defects in the system and to reveal problems
- Contrary to all other activities like analysis, design or implementation: testing is not constructive

→ Design: avoid making faults

→ A successful test is a test that identifies faults

Testing (ctd.)

- Alternative Definition: Testing has to demonstrate that faults are not present at all.
  - Almost impossible to show
  - May lead to the selection of test data that have a low probability of causing the program to fail

- Many definition of “errors” can be found ...

Terminology

- **Fault** (Bug): A design or coding mistake that may cause abnormal component behavior.
  - Algorithmic fault: caused by wrong implementation of the specification (mostly due to bad communication)
  - Mechanical fault: due to external circumstances
- **Error**: The system is in a state such that further processing by the system will lead to a failure.
- **Failure**: Any perceivable deviation of the observed behavior from the specified behavior.

- There are many different ways how we can deal with these types of errors.

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
  - Missing test for null or 0
- Mechanical Faults
  - very hard to find
  - Documentation does not describe actual conditions of environment
- Errors
  - Stress or overload errors
  - Capacity or boundary errors
  - Timing errors
  - Throughput or performance errors

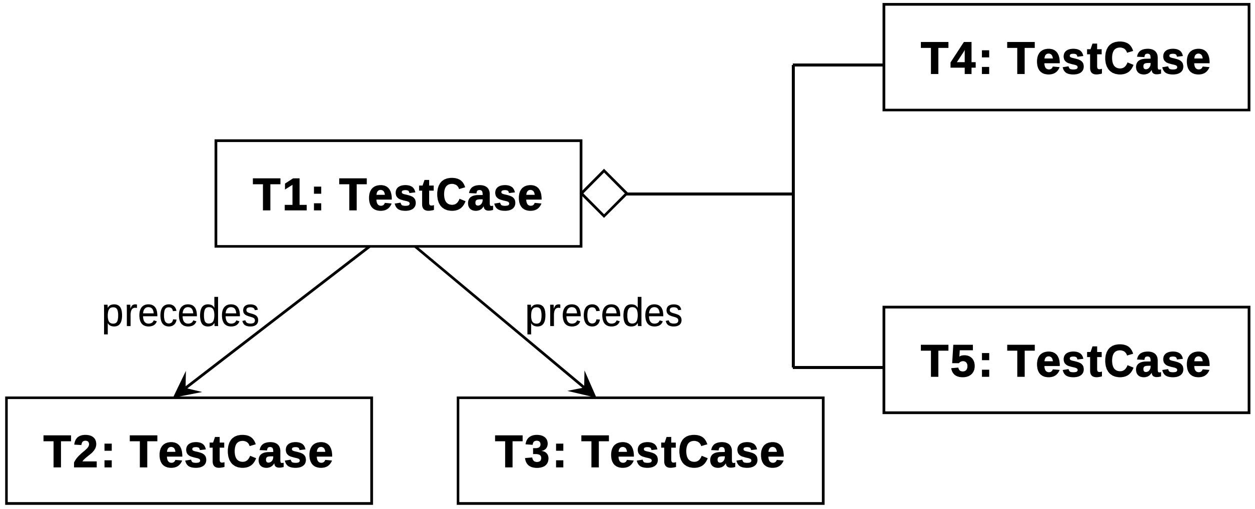
➔ **How do we deal with Errors and Faults?**

Dealing with Errors

- Verification
  - Formal proof of correctness.
  - Assumes hypothetical environment that does not match real environment
  - Proof might be buggy (omits important constraints; may be simply wrong)
- Declaring a bug to be a “feature”
  - Bad practice ☹
- Patching
  - Rather quick and dirty...
- Testing (this lecture)
  - Testing is never good enough
  - Define Test Cases even during all stages in order to detect faults

Test Cases

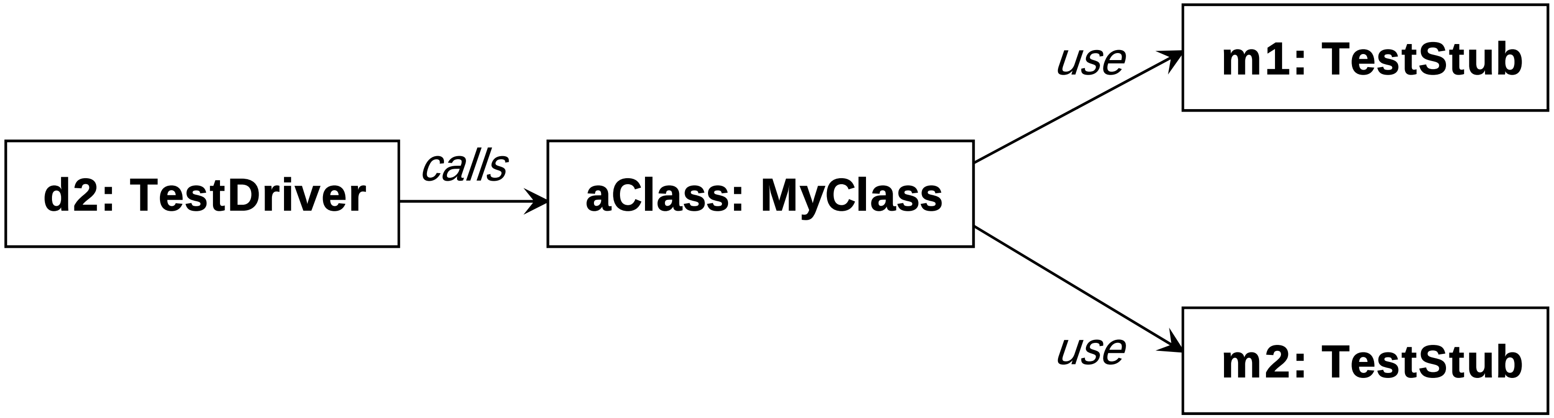
- **Test Case:** set of input data and expected results that exercise a component with the purpose of causing failures and detecting faults
- Test cases can have relationships:



- Expected results are sometimes called **Test Oracle.**

Test Stubs and Drivers

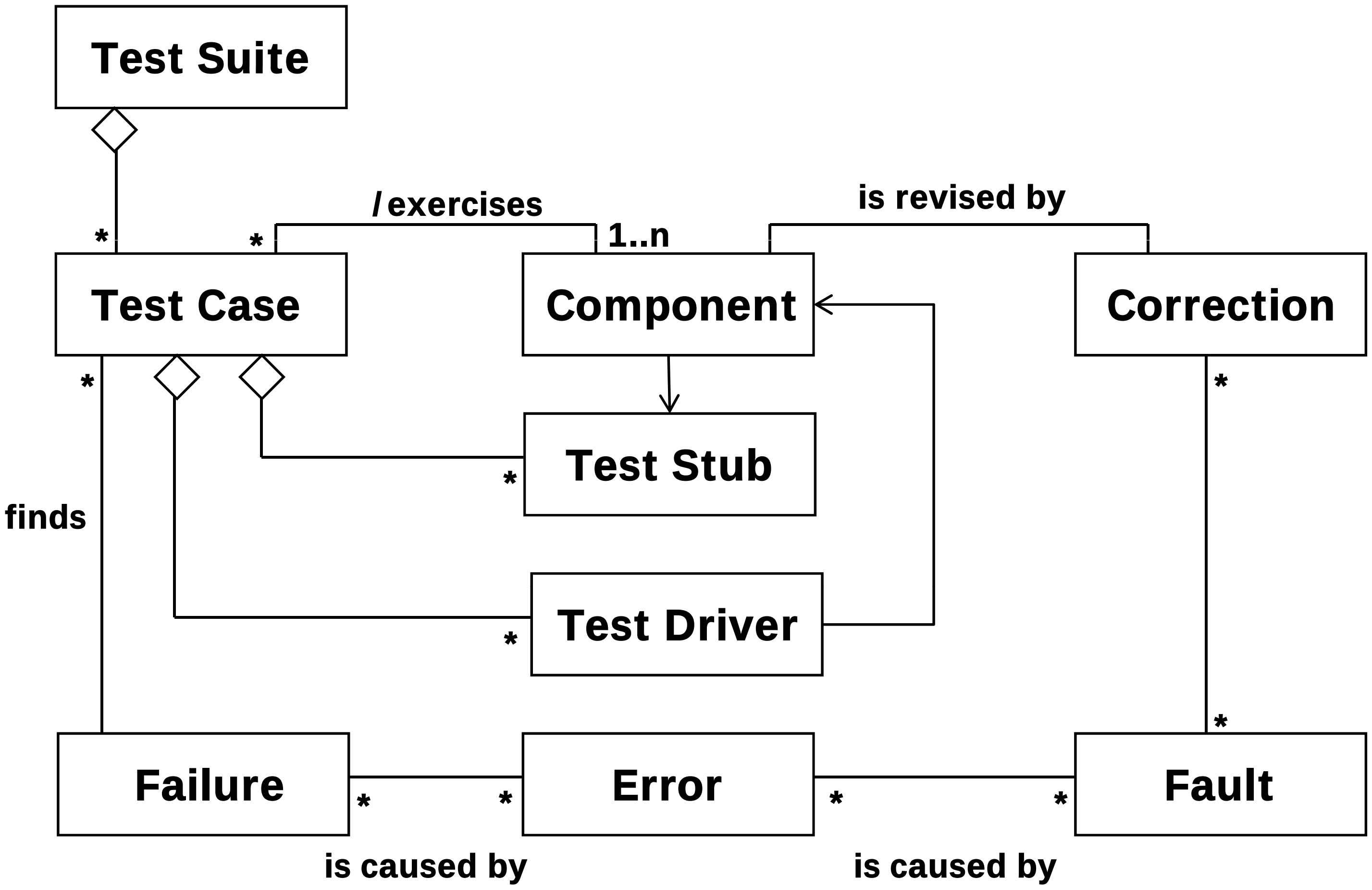
- **Test Driver:** simulates the part of the system that calls the **component under test (CUT)**
- **Test Stub (or Mock):** simulates component that are called by the tested component
  - Must provide the same API as the intended component
  - Not always a trivial task



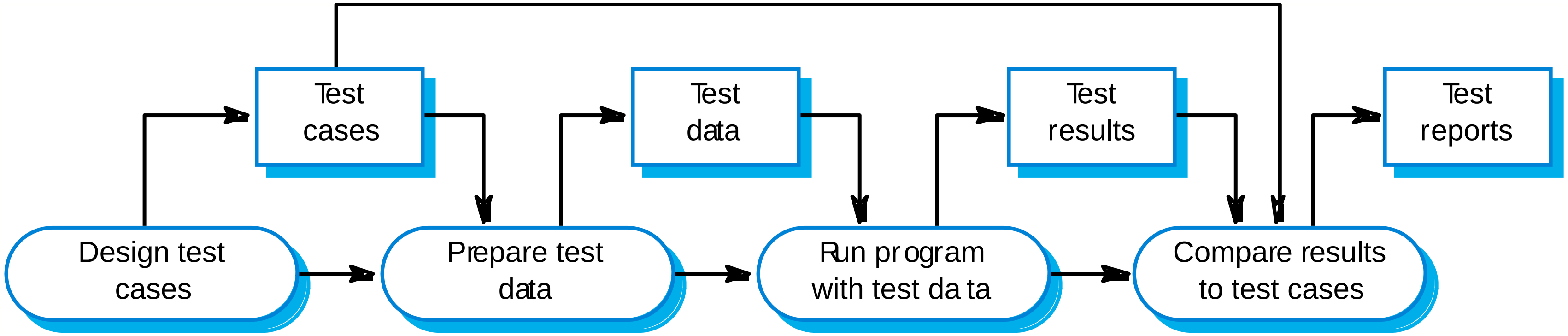
Corrections

- A Correction is a change to a component whose purpose is to repair a fault.
  - Range from simple modification to a single component, to a complete redesign of data structures or a subsystem
- The likelihood that the developer introduces new faults into the revised component is high. Techniques to handle such faults:
  - Configuration Management
  - Rationale Management (Documentation of the rationale for the change)
- “Detecting and fixing one bug naturally causes five new bugs..” (Source: unknown, ~Fragility, R.C. Martin)

Model elements used during test



The software testing process



Testing takes creativity

- To develop an effective test, one must have:
    - Detailed understanding of the system
    - Knowledge of the testing techniques
    - Skill to apply these techniques in an effective and efficient manner
- Testing is done best by independent testers
    - Developers often develop a mental attitude that the program should behave in a certain way when in fact it does not.
    - Programmer often stick to the data sets that makes the program work
- A program often does not work when first tried by *somebody* else.
    - Don't let this be the end-user or client.

Types of Testing

- Unit Testing: **today**
  - Individual subsystem
  - Carried out by developers (of components)
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface among the subsystem

- Implementation (Coding) and testing go hand in hand

Types of Testing

- System Testing:
  - The entire system
  - Carried out by test team
  - Goal: Determine if the system meets the requirements (functional and global)
  - Functional Testing: Test of functional requirements
  - Performance Testing: Test of non-functional requirements

Types of Testing

- Acceptance and Installation Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client.
  - Goal: Demonstrate that the system meets customer requirements and is ready to use

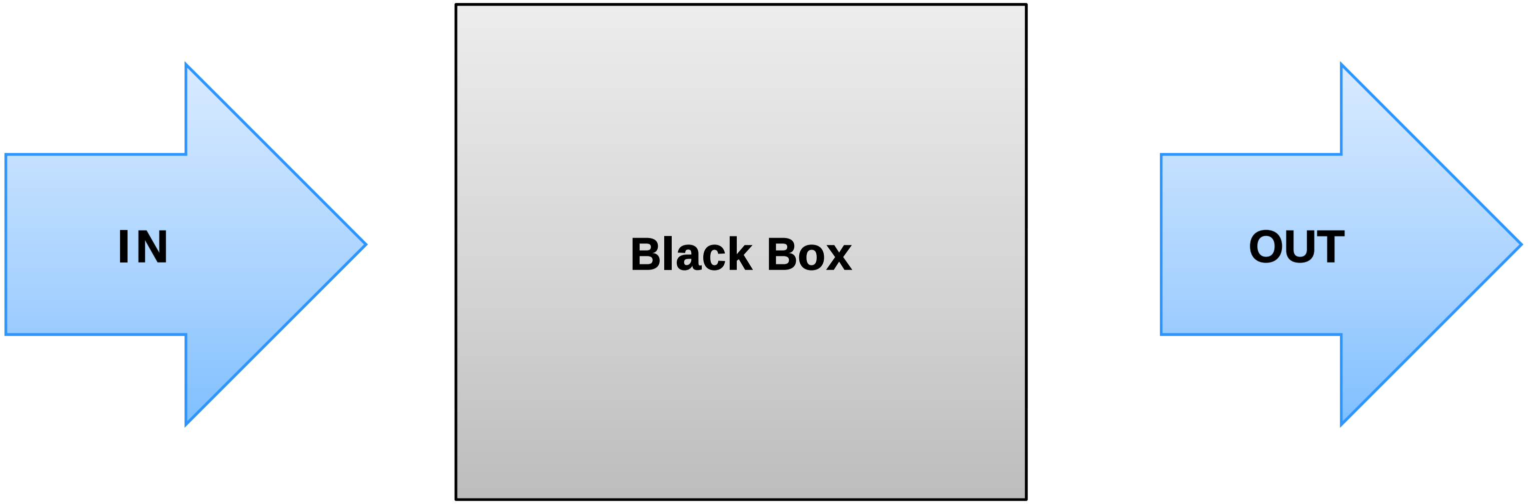
Unit Testing

- Focus on the building blocks of the software: objects and subsystems. Benefits:
  - Reduction on complexity of overall test activities
  - Makes it easy to detect and correct faults
  - Write test once and use it to find errors many times

- All objects developed during development can be involved
  - Often not feasible (and necessary)
  - Minimal set: participating objects of analysis phase (In order of importance: Entities, Controller, and maybe Boundaries)
  - Subsystem should be tested after all objects within the subsystem have been tested individually

Unit Testing

- Static Analysis:
    - Code Review: Reading the source code
    - Walkthrough (Informal presentation of code and API to review team)
    - Inspection (No involvement of developers)
    - Automated Tools checking for
        - syntactic errors, coding standards
- Dynamic Analysis:
    - Black-box testing (Test the input/output behavior)
    - White-box testing (Test the internal logic of the subsystem or object)



IN

Black Box

OUT

Test Data

Expected Data

Actual Data

Ideally: All possible value
Unmanagable! To expensive!
=> Equivalence classes

Comparison of expected and actual data

Equivalence Classes, Examples

- Square Root
  - Negative, Zero, Positive; Natural, Rational, Irrational root
  - Test data = {-16, 0, 25, 16/25, 7}
  - Expected Result = {4, 0, 5, 0.8, 2.64575131}

- Greatest Common Divisor
  - (1,a), (a,a), (p,q),
  - (p*a, p), (a*p, a*q), (p*q, r*s)
  - Test data =
    {(1, 8), (23, 23), (7, 11), (22, 11), (14, 22), (3*7, 11*2)}
  - Expected Result = {1, 23, 1, 11, 2, 1}

Black-Box Testing

- Focus: I/O behavior. If for any given input we can predict the output, then the module passes the test.
  - Do not deal with the internal aspects of the tested component
  - Almost always impossible to generate all possible inputs
- Goal: Reduce number of test cases
- Method: Equivalence Testing
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-Box Testing (Continued)

- Boundary testing:
  - Focus on the conditions at the boundary (edges) of the equivalence classes
  - Select test cases from 3 equivalence classes:
    - Below the range (e.g. 0, null)
    - Within the range (any number of String)
    - Above the range (huge number of big Strings)
- Disadvantage (Equivalence and Boundary Testing):
  - Do not explore combinations of test input data
  - Often, a combination of certain values causes the erroneous state
- Another solution to select only a limited amount of test cases:
  - Get knowledge about the inner workings of the unit being tested => white-box testing

White-Box Testing

- Focus on the internal structure of the component.
- **Goal**: each state in dynamic model of an object and each interaction among the objects should be tested.
- Four quality metrics for white-box testing:
  - Statement Coverage
    - Is each statement exercised (covered) by a test?
  - Loop Coverage
    - Is each loop body executed zero times, exactly once, and more than once (consecutively)?
  - Branch Coverage
    - Is each possible outcome of an decision covered?
  - Path Coverage
    - Is each possible path covered?

White-Box Testing
Path Testing

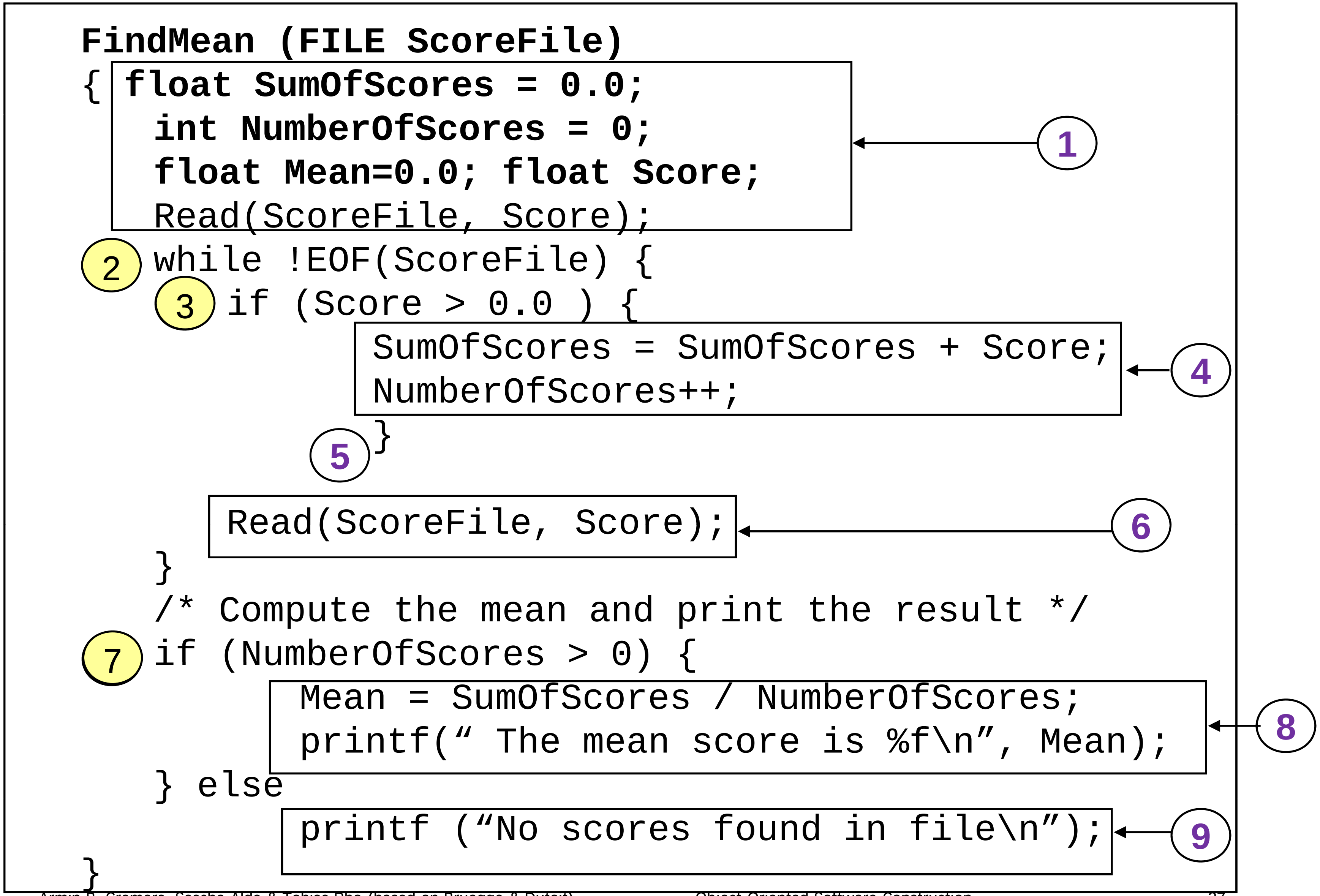
- Assumption: by exercising all paths through a code, most faults will trigger failures
- Make sure all paths in the program are executed
- Make sure that each possible outcome from a condition is tested at least once

  -
    ```
    if (i == TRUE) out.writeln("YES");
    else out.writeln("NO");
    ```
  - Test cases: 1) i = TRUE; 2) i = FALSE
- Starting Point for more complex code fragments: flow graphs
  - Nodes: executable blocks
  - Association: representing decision statement (if, while)

White-Box Testing Example

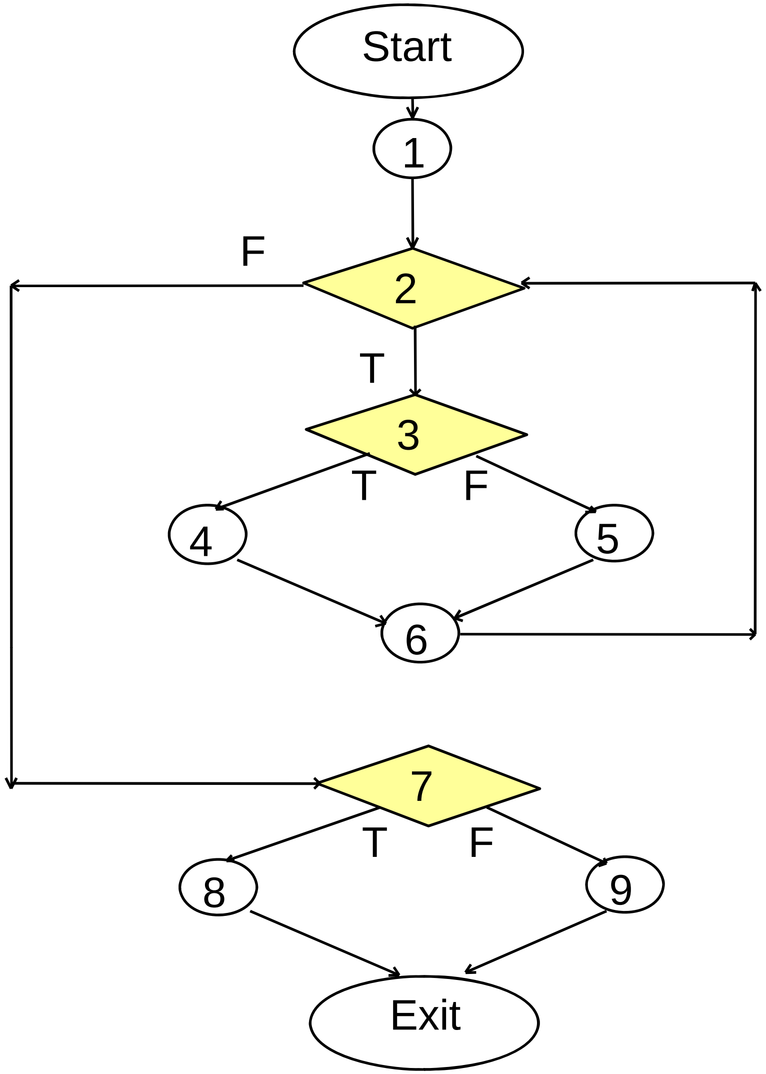
```
FindMean (FILE ScoreFile)
{ float SumOfScores = 0.0;
   int NumberOfScores = 0;
   float Mean=0.0; float Score;
   Read(ScoreFile, Score);
   while !EOF(ScoreFile) {
       if (Score > 0.0 ) {
             SumOfScores = SumOfScores + Score;
             NumberOfScores++;
             }

       Read(ScoreFile, Score);
   }
   /* Compute the mean and print the result */
   if (NumberOfScores > 0) {
          Mean = SumOfScores / NumberOfScores;
          printf(“ The mean score is %f\n”, Mean);
   } else
          printf (“No scores found in file\n”);
}
```

White-Box Testing Example: Determining the Paths



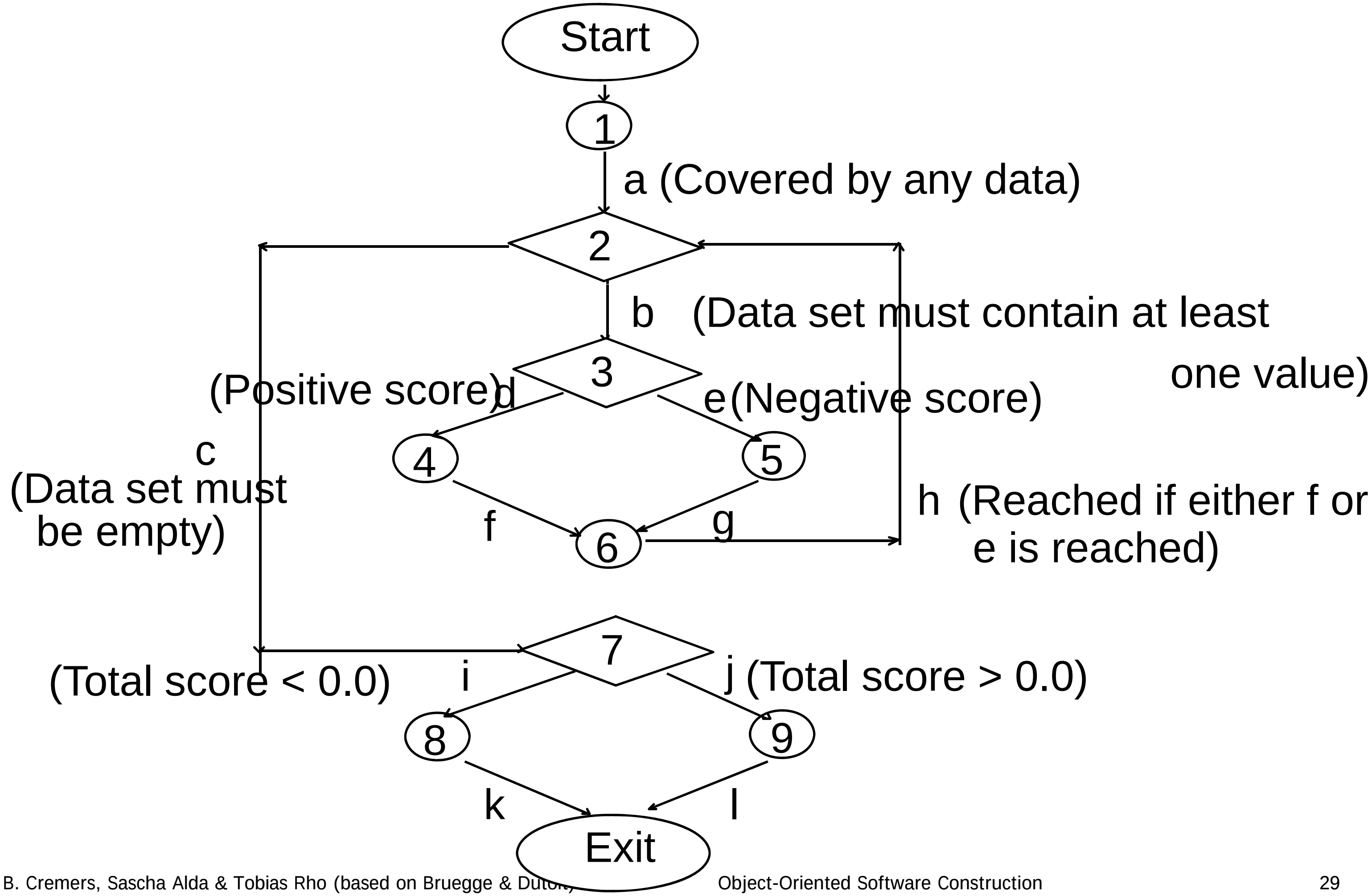
```
FindMean (FILE ScoreFile)
{ float SumOfScores = 0.0;
    int NumberOfScores = 0;
    float Mean=0.0; float Score;
    Read(ScoreFile, Score);
    while !EOF(ScoreFile) {
        if (Score > 0.0 ) {
                SumOfScores = SumOfScores + Score;
                NumberOfScores++;
                }

        Read(ScoreFile, Score);
    }
    /* Compute the mean and print the result */
    if (NumberOfScores > 0) {
            Mean = SumOfScores / NumberOfScores;
            printf(" The mean score is %f\n", Mean);
    } else
            printf ("No scores found in file\n");
}
```

Constructing the Logic Flow Diagram



Finding the Test Cases

- Design test cases so that each transition in the activity diagram is traversed at least once
  - select input for true and false branch



Unit Testing in Java

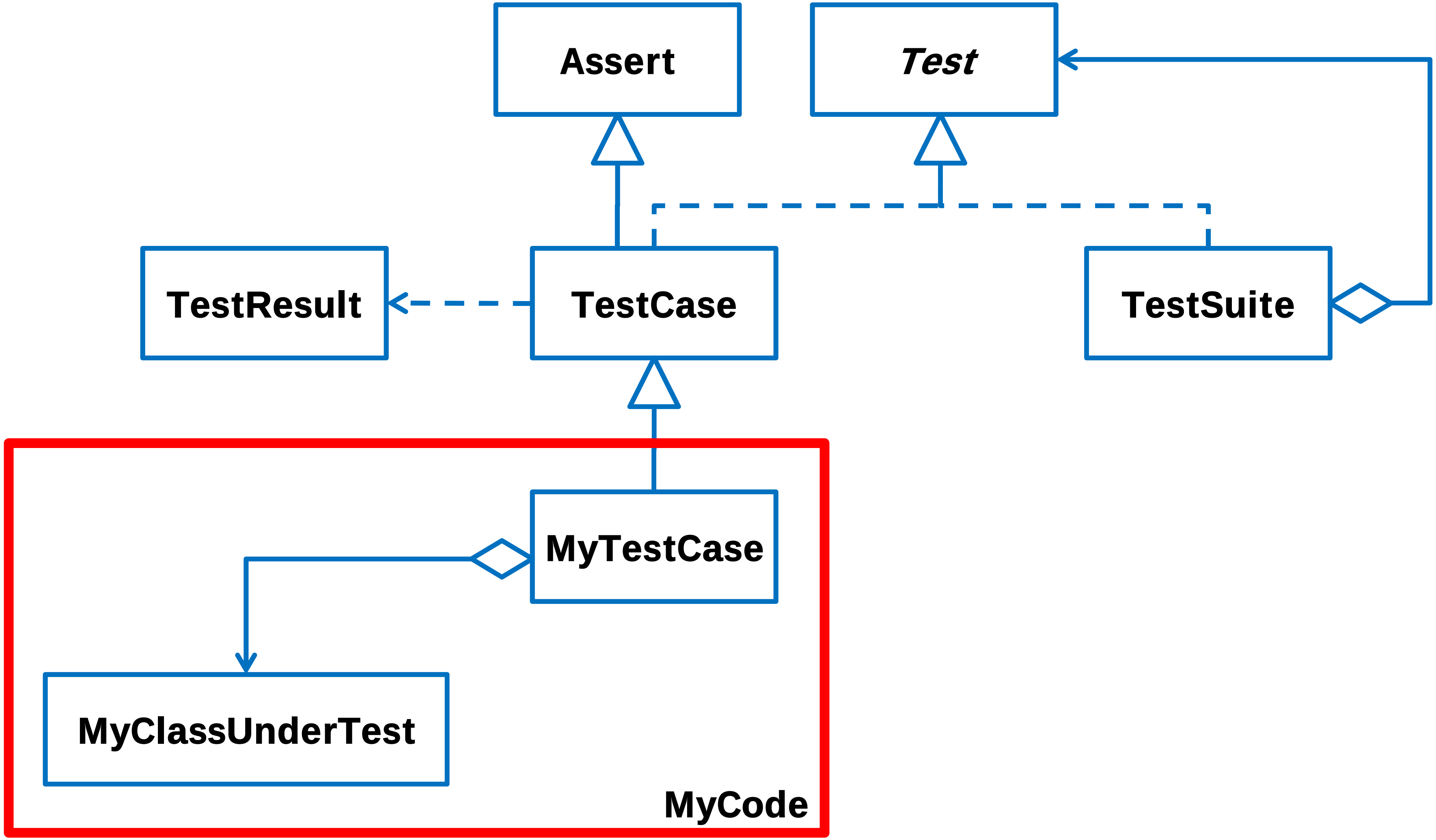
- (Trivial) Testing of single Objects:
  - Build up Test Cases by means of additional main() method that invokes individual methods.
  - Use of System.out.println() command to check values
  - Advantages:
    - Very easy to use and insert
  - Disadvantages:
    - Annoying code in the business code
    - Too many unnecessary outputs
    - Test code is interweaved with business code (no portability)

Testing in Java JUnit

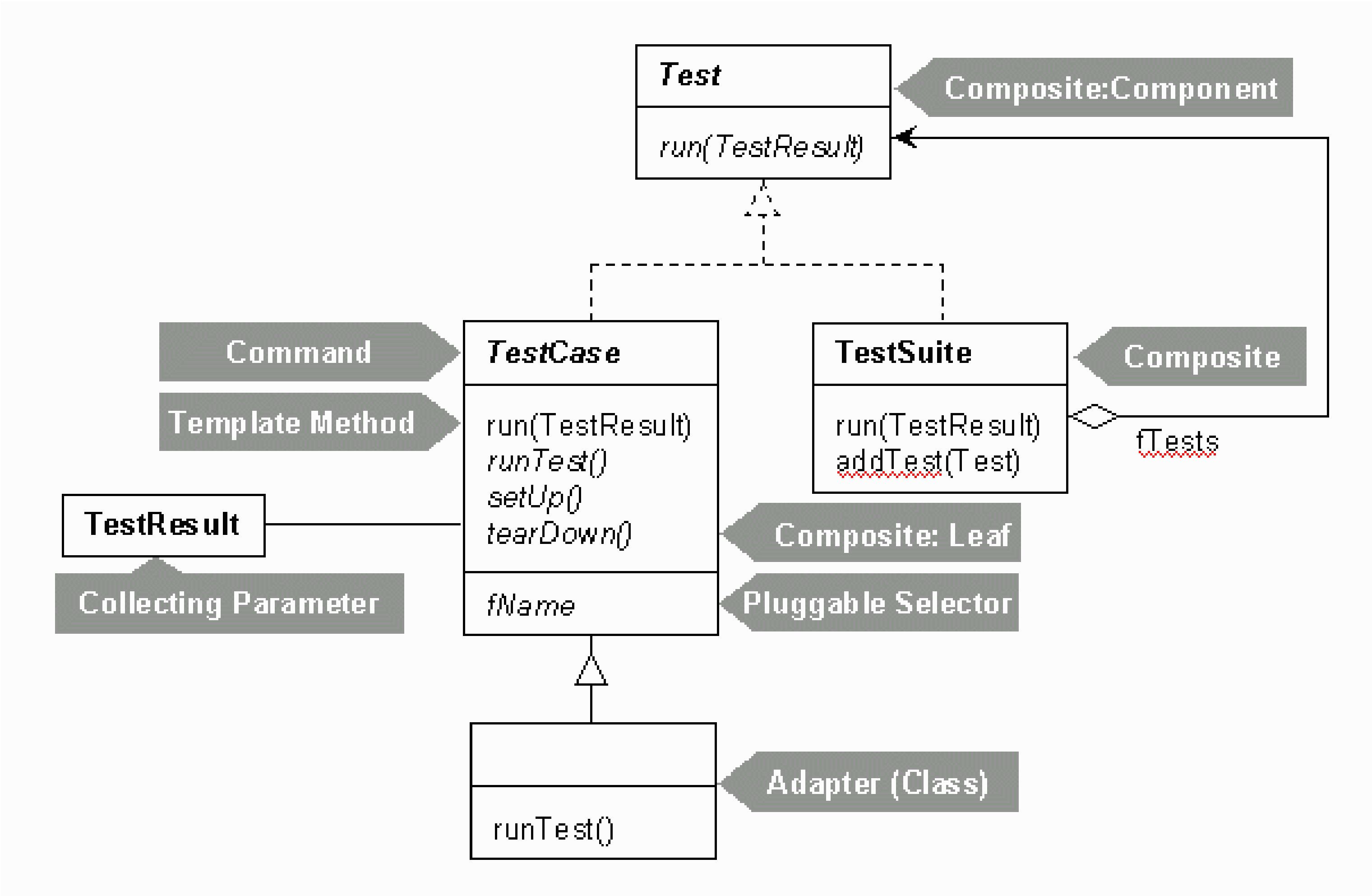
- De facto standard Java framework for unit (object) testing
- Realization of real TestCases and TestSuites
- TestCases are easily portable to other units
- Separation of test code and business code of object
- Integrated nicely with existing IDEs like Eclipse

- Use of arbitrary assertions to evaluate values

JUnit Design



JUnit Design - Pattern dense



http://junit.sourceforge.net/doc/cookstour/cookstour.htm

JUnit Rules and Conventions

- Subclass **`TestCase`**
- Test methods
  - **`public void test`*`XXX`*`() [throws …]`**
  - Any number of assertions per method

- Optionally add setUp / tearDown methods
  - Instantiating (auxiliary) objects
  - Network setups
  - Integration of Mock-Up Objects (Test Stubs)

Example Code

```
package org.example.antbook.common;

public class SearchUtil {

    public static final Document[]
                    findDocuments(String queryString)
                         throws SearchQueryException,
                                SystemException {
        Document[] results = new Document[1];
        return results;
    }
}
```

- Test: what is the size of results?
- Does the method really returns a document?

An example unit test

```java
package org.example.antbook.common;

import junit.framework.TestCase;

public class SearchUtilTest extends TestCase {

    public void testSearch() throws Exception {
        // right API?
        Document[] docs =
                   SearchUtil.findDocuments("erik");

        assertTrue(docs.length > 0);
    }
}
```

JUnit Assertions

- assertTrue(boolean condition)
  assertFalse(boolean condition)

- assertEquals(Object expected, Object actual)
  - Uses equals() comparison (check whether two object have the same content)

- assertSame(Object expected, Object actual)
  assertNotSame(Object expected, Object actual)
  - Uses == comparison (check if two objects refer to the same object)

- assertEquals(float expected, float actual, float tolerance)

- assertNull(Object o)
  assertNotNull(Object o)

Test Runners

- Execute unit test via command line:

`java junit.USERINTERFACE.TestRunner` ***`classfile`***

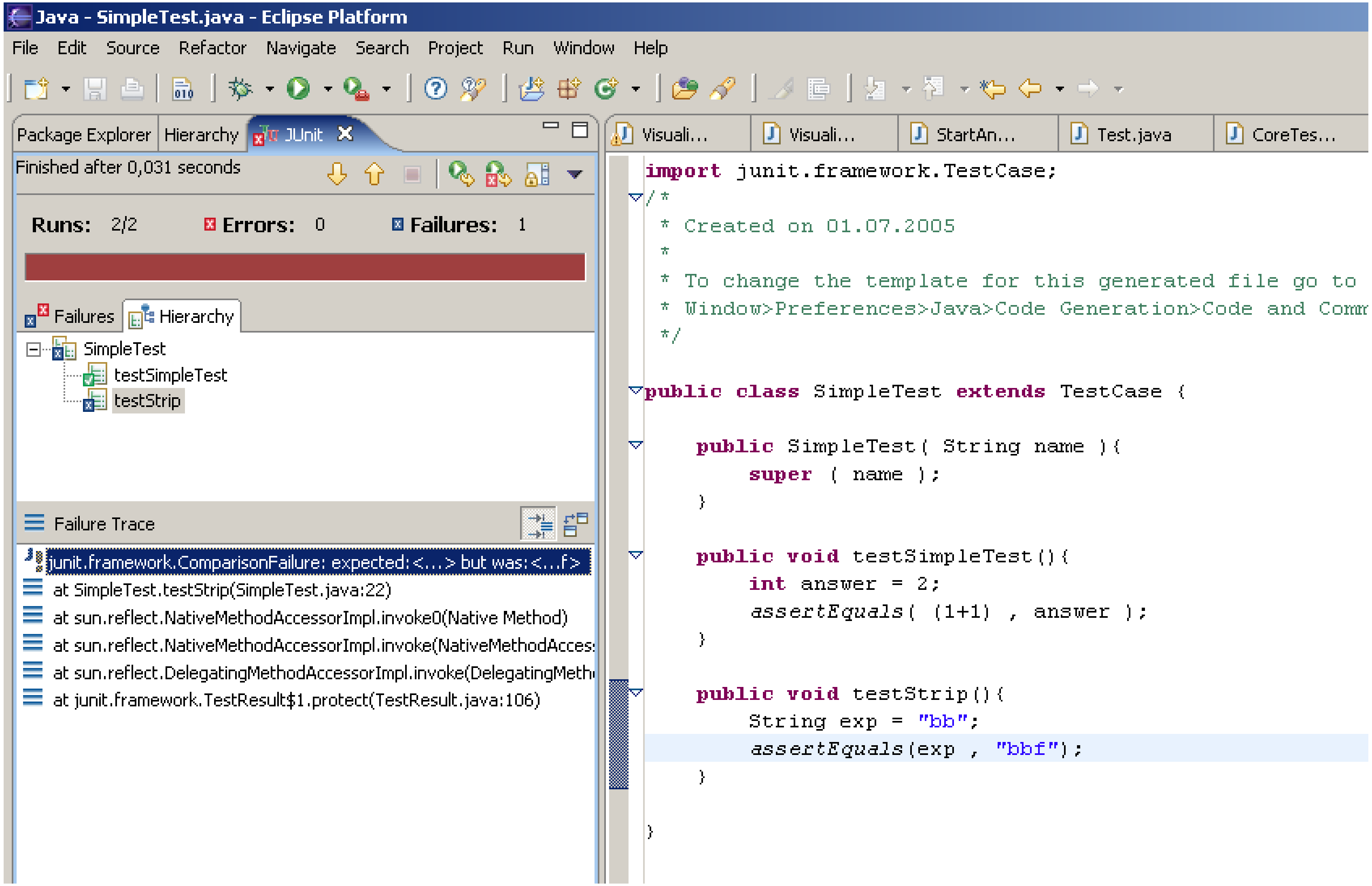
- The UserInterface package prescribes the output style for the computed result: It can hold one of the following values:
  - textui (textual representation of the result)

```
> java junit.textui.TestRunner SearchUtilTest

>
> Time: 0
> OK (1 tests)
```

  - SwingUI (graphical representation using Swing components)
  - AwtUI (graphical representation using Awt components)

- Better: Use Eclipse ...

JUnit in Eclipse



Finished after 0,031 seconds

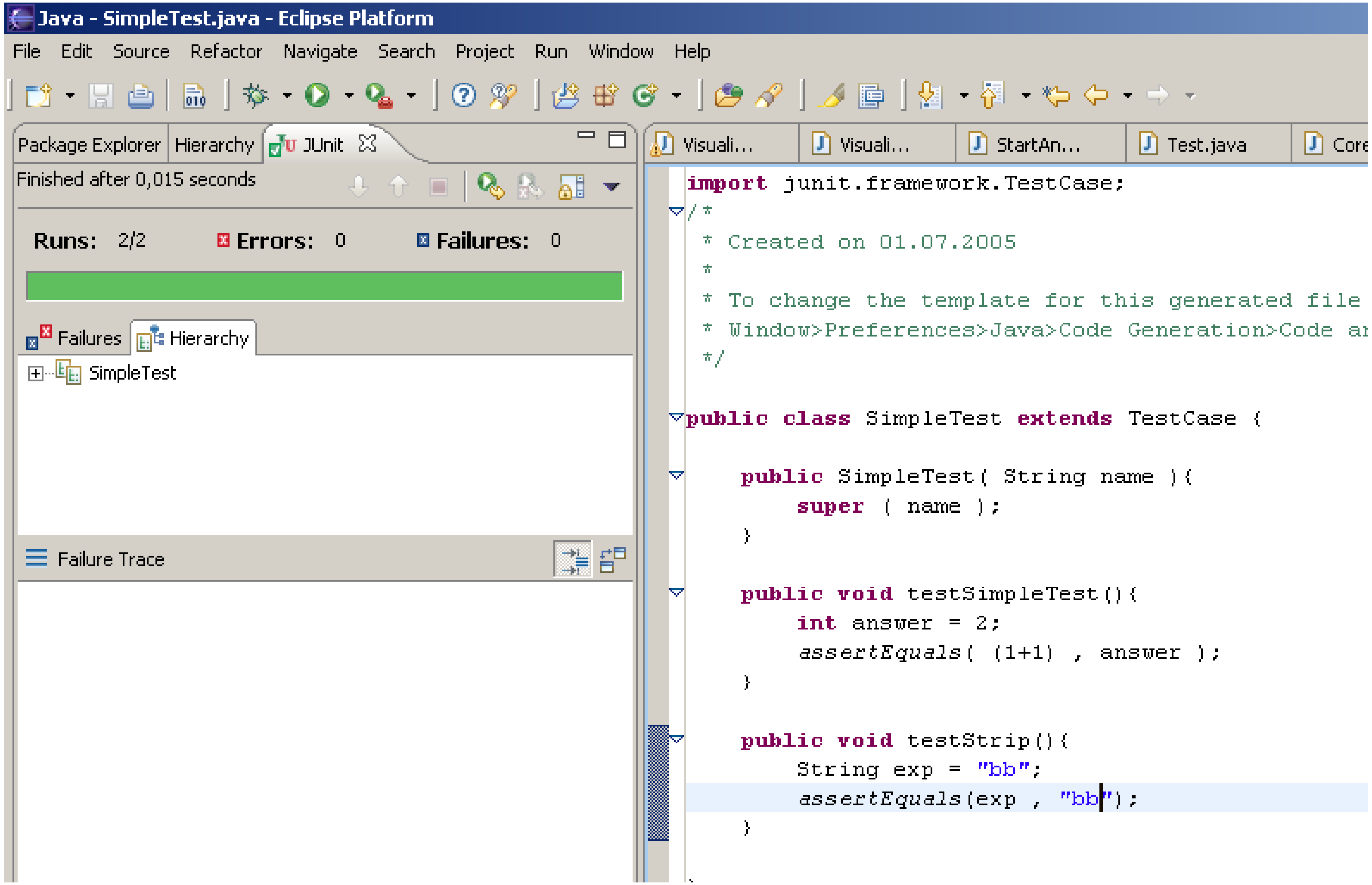
Runs: 2/2 ☒ Errors: 0 ☒ Failures: 1

- SimpleTest
  - testSimpleTest
  - testStrip

```
junit.framework.ComparisonFailure: expected:<...> but was:<...f>
at SimpleTest.testStrip(SimpleTest.java:22)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMeth
at junit.framework.TestResult$1.protect(TestResult.java:106)
```

```
import junit.framework.TestCase;
/*
 * Created on 01.07.2005
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comm
 */

public class SimpleTest extends TestCase {

    public SimpleTest( String name ){
        super ( name );
    }

    public void testSimpleTest(){
        int answer = 2;
        assertEquals( (1+1) , answer );
    }

    public void testStrip(){
        String exp = "bb";
        assertEquals(exp , "bbf");
    }

}
```

JUnit in Eclipse



Finished after 0,015 seconds

Runs: 2/2 Errors: 0 Failures: 0

```
import junit.framework.TestCase;
/*
 * Created on 01.07.2005
 *
 * To change the template for this generated file
 * Window>Preferences>Java>Code Generation>Code ar
 */

public class SimpleTest extends TestCase {

    public SimpleTest( String name ){
        super ( name );
    }

    public void testSimpleTest(){
        int answer = 2;
        assertEquals( (1+1) , answer );
    }

    public void testStrip(){
        String exp = "bb";
        assertEquals(exp , "bb");
    }
```

Lifecycle Methods

```
package org.example.antbook.ant.lucene;

import java.io.IOException;
import junit.framework.TestCase;

public class HtmlDocumentTest extends TestCase
{
    HtmlDocument doc;

    public void setUp() throws IOException {
        doc = new HtmlDocument(getFile("test.html"));
    }

    public void testDoc() {
        assertEquals("Title", "Test Title", doc.getTitle());
        assertEquals("Body", "This is some test", doc.getBodyText());
    }

    public void tearDown() {
        doc = null;
    }
}
```

TestCase lifecycle

1. **`setUp`**
2. **`test`*`XXX`*`()`**
3. **`tearDown()`**
4. Repeats 1 through 3 for each **`test`*`XXX`*** method...

Test Suites

```
package org.example.antbook;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class AllTests {

public static void main(String[] args) {
    junit.textui.TestRunner.run( AllTests.class );
 }
public static public Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(SimpleTest.class);
    suite.addTestSuite(HtmlDocumentTest.class);
    return suite;
  }
}
```

Use of Reflection

JUnit Best Practices

- Separate business and test code
- But typically in the same packages
- Compile into separate trees, allowing deployment without tests
- Don’t forget OO techniques

- Test-driven development
  1. Write failing test first
  2. Write enough code to pass
  3. Refactor code
  4. Run tests again
  5. Repeat until software meets goal

Summary

- Testing still needs intuition, but many rules and heuristics are available
- Testing consists of component-testing (unit testing, integration testing) and system testing
- Design Patterns can be used for integration testing
- Testing has its own lifecycle