

eXtreme Programming

(summary of Kent Beck's XP book)

Prof. Dr. Wolfgang Pree
Universität Salzburg
pree@SoftwareResearch.net

Contents

- The software development problem
- The XP solution
- The JUnit testing framework

The SW development problem

Four variables

Overview

- **cost**
- **time**
- **quality**
- **scope**

**external forces (customers,
management) pick the values of 3 v.
solution: make the four variables visible**

interaction between the variables

- time: more time can improve quality and increase scope
too much time will hurt it
- quality: short-term gains by deliberately sacrificing quality; but the cost (human, business, technical) is enormous
- less **scope** => better quality (as long as the business problem is still solved)

Four values

Overview

- **communication**
- **simplicity**
- **feedback**
- **courage**

short-term vs. long term thinking (I)

- communication: effect of pair programming, unit testing, task estimation: programmers, customers and managers have to communicate
- simplicity: it is better to do a simple thing today and pay a little more tomorrow to change it if it needs than to do a more complicated thing today that may never be used anyway

short-term vs. long term thinking (II)

- feedback: when customers write new „stories“ (description of features, simplified use cases), the programmers immediately estimate them; customers and testers write functional tests for all the stories
- courage: throwing parts of the code away and start over on the most promising design

Basic principles (derived from the four values)

Basic principles (I)

- **rapid feedback**
- **assume simplicity**
- **incremental change**
- **embracing change**
- **quality work**

Basic principles (II)

- **small initial investment**
- **play to win**
- **concrete experiments**
- **open, honest communication**
- **work with people's instincts, not against them**

Basic activities

Basic activities in the XP development process

- coding
- testing
- listening
- designing

The solution

XP practices

Practices (I)

- **planning game: determine the scope of the next release; as reality overtakes the plan update the plan**
- **small releases: release new versions on a very short cycle after putting a simple system into production quickly**
- **metaphor: guide development with a simple shared story of how the whole system works**

Practices (II)

- **simple design: as simple as possible but not simpler (A. Einstein)**
- **testing: continually write unit tests**
- **refactoring: restructure the system to remove duplication (c.f. framelets, etc.)**
- **pair programming: two programmers at one machine**
- **collective ownership**

Practices (III)

- **continuous integration: integrate the system many times a day, every time a task is complete**
- **40-hour week**
- **on-site customer: include a real, live customer**
- **coding standards**

Management strategy

Overview

- **decentralized decision making based on**
 - metrics
 - coaching
 - tracking
 - intervention
- **using business basics: phased delivery, quick and concrete feedback, clear articulation of the business needs, specialists for special tasks**

Metrics

- **don't have too many metrics**
- **numbers are regarded as a way of gently and noncoercively communicating the need for change**
- **ratio between the estimated development time and calendar time is the basic measure for running the Planning Game**

Coaching

- **be available as a development partner**
- **see long-term refactoring goals**
- **explain the process to upper-level management**

=> no lead programmer, system architect, etc.

Intervention

- **when problems cannot be solved by the emergent brilliance of the team, the manager has to step in, make decisions and see the consequences through to the end**
- **sample situations: changing the team's process, personnel changes, quitting a project**

Planning strategy

Overview

- **bring the team together**
- **decide on scope and priorities**
- **estimate cost and schedule**
- **give everyone confidence that the system can be done**
- **provide a benchmark for feedback**

put the most valuable functionality into production asap

Summary

What makes XP hard?

It's hard to ...

- do simple things
- admit you don't know (eg, basics about computer/software science in the context of pair programming)
- to collaborate
- to break down emotional walls

XP & Kent Beck (I)

Kent Beck is afraid of:

- **doing work that doesn't matter**
- **having projects canceled**
- **making business decisions badly**
- **doing work without being proud of it**

XP & Kent Beck (II)

Kent Beck is not afraid of:

- **coding**
- **changing his mind**
- **proceeding without knowing everything about the future**
- **relying on other people**
- **changing the analysis and design of a running system**
- **writing tests**

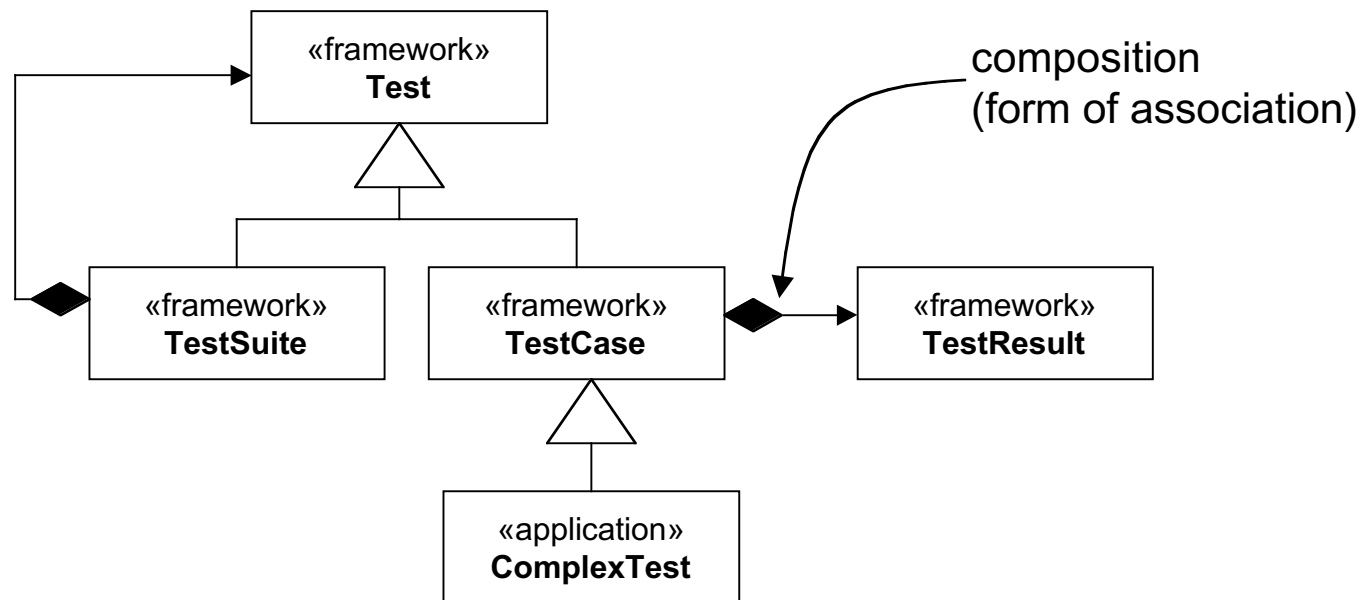
The JUnit testing framework

The JUnit components (I)

- Adding new test cases: JUnit provides a standard interface for defining test cases and allows the reuse of common code among related test cases.
- Tests suites: Framework users can group test cases in test suites.
- Reporting test results: the framework keeps flexible how test results are reported. The possibilities include storing the results of the tests in a database for project control purposes, creating HTML files that report the test activities.

The JUnit components (II)

Overview of the JUnit design - Class
ComplexTest defines test cases for
complex numbers

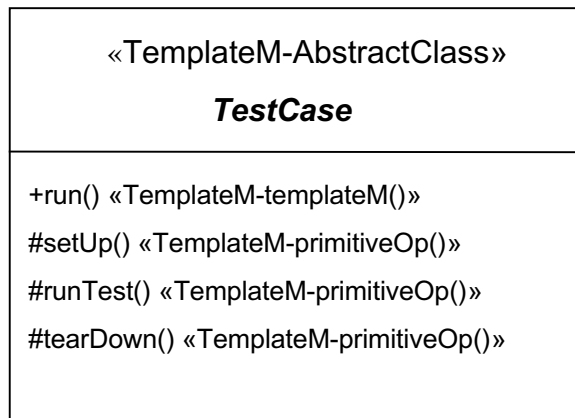


The TestCase variation point (I)

- The initialization part is responsible for creating the test fixture.
- The test itself uses the objects created by the initialization part and performs the actions required for the test.
- Finally, the third part cleans up a test.

The TestCase variation point (II)

The TestCase design is based on the Template Method design pattern - method `run()` controls the test execution

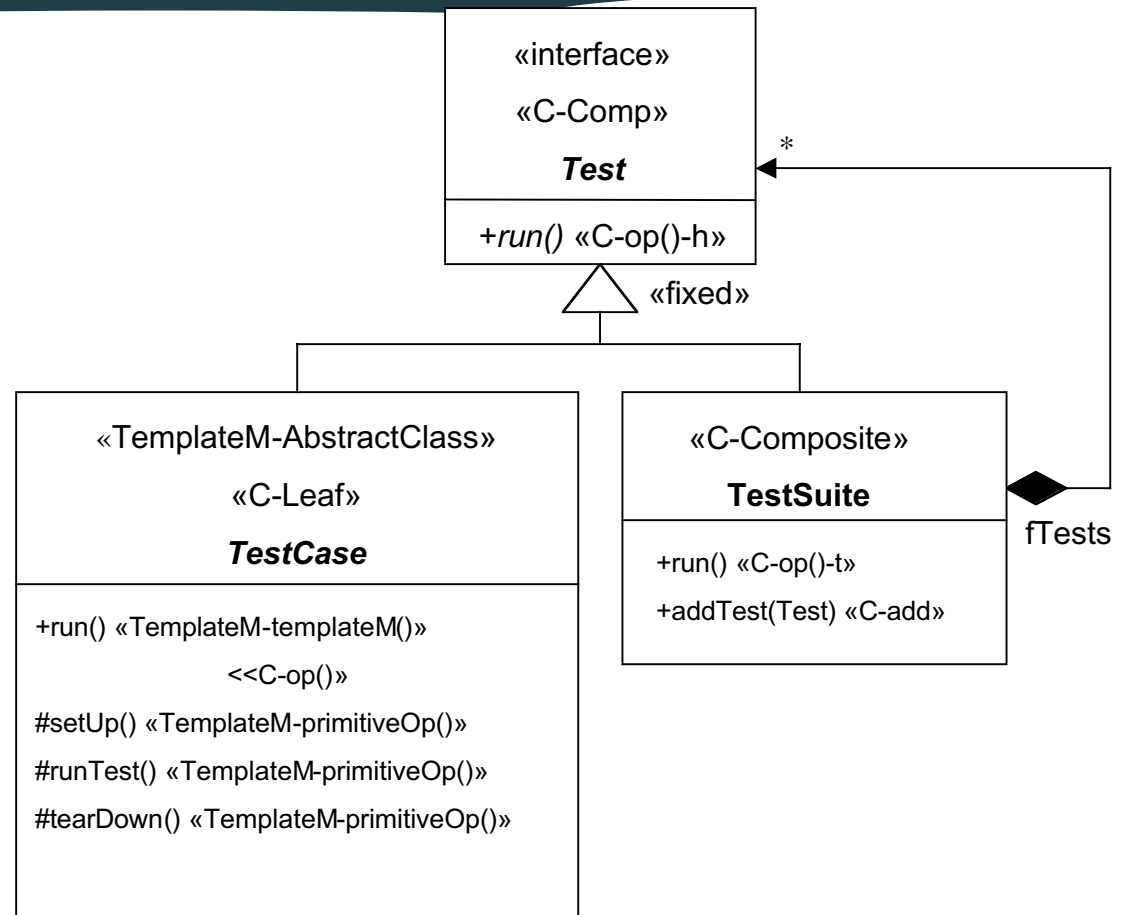


```
public void run() {  
    setUp();  
    runTest();  
    tearDown();  
}
```

The TestSuite variation point

TestCases are grouped into TestSuites—a variation of the Composite design pattern

Black-box adaptation

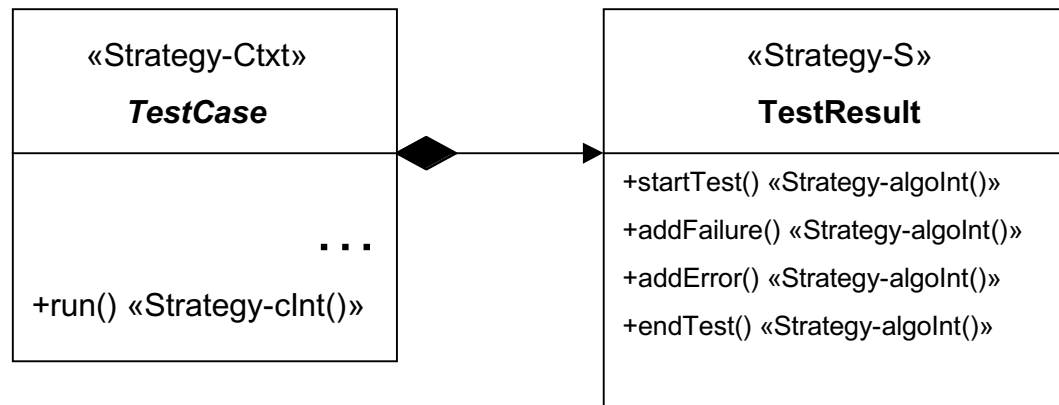


The TestResult variation point (I)

- Failures are situations where the assert() method does not yield the expected result.
- Errors are unexpected bugs in the code being tested or in the test cases themselves.
- The TestResult class is responsible for reporting the failures and errors in different ways.

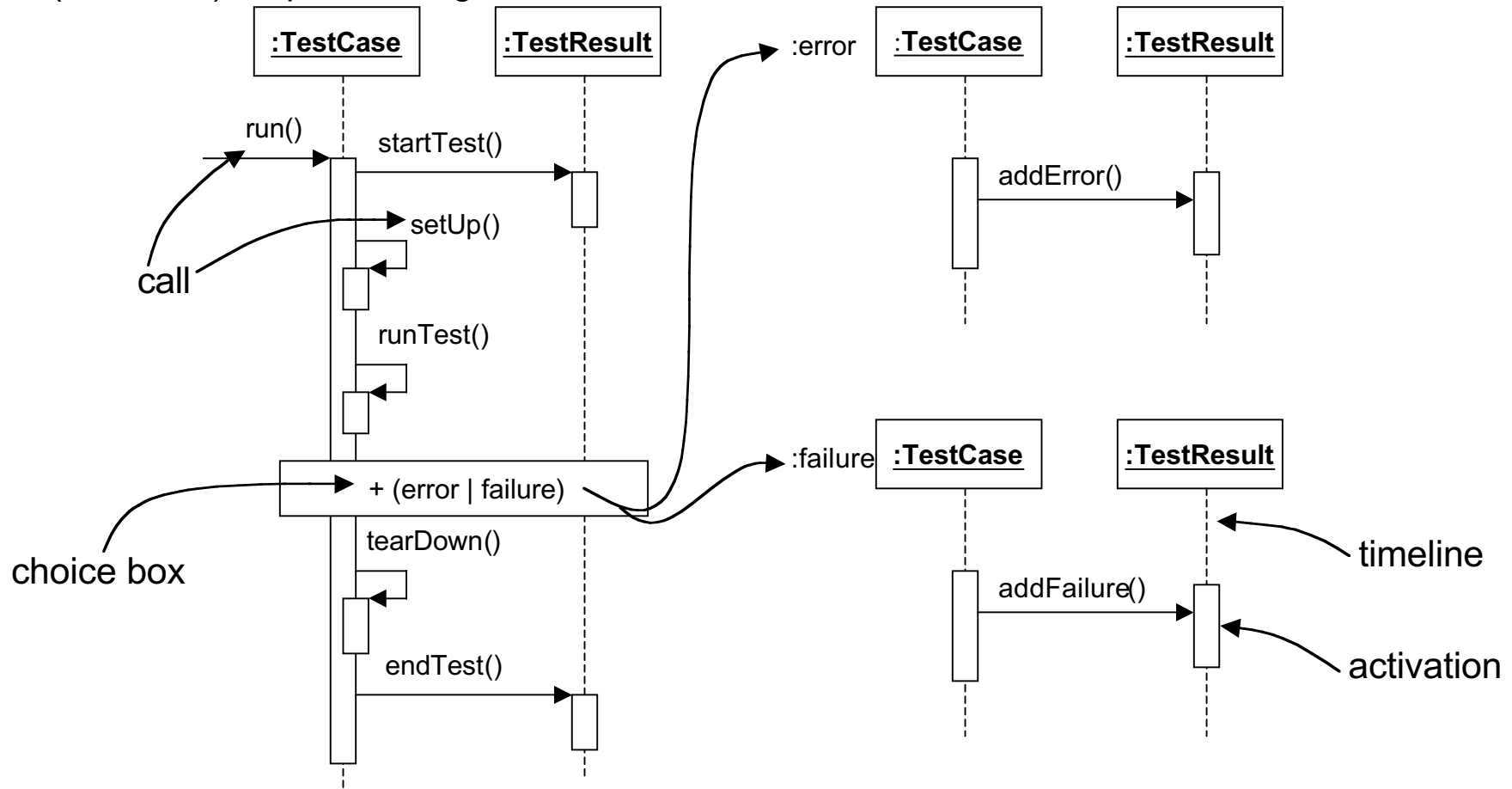
The TestResult variation point (II)

- TestResult must provide four methods:
 - startTest() - initialization code
 - addFailure() - reports a failure
 - addError() - reports an error
 - endTest() - clean-up code



The TestResult variation point (III)

(extended) sequence diagram



Adapting JUnit

- Cookbook recipes and UML-F diagrams for each of the JUnit variation points
 - Create a test case (ComplexTest)
 - Create a test suite (for the ComplexTest methods)
 - Create an HTML reporting mechanism

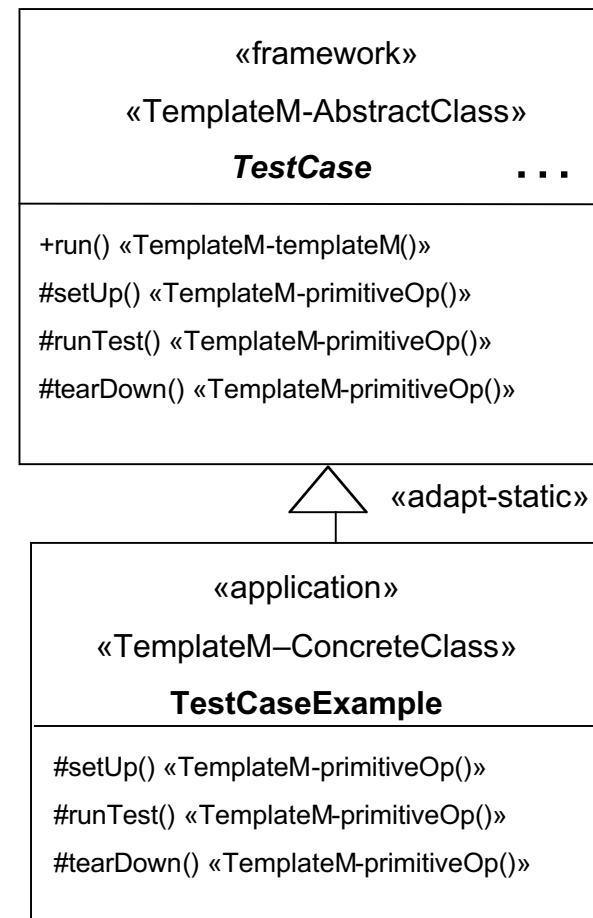
Adapting TestCase (I)

- TestCase adaptation recipe:
 - Subclass TestCase
 - Override setUp() (optional). The default implementation is empty
 - Override runTest()
 - Override tearDown() (optional). The default implementation is empty

Adapting TestCase (II)

TestCaseExample exemplifies the code that has to be added by the application developer

White-box adaptation



Adapting TestCase (III)

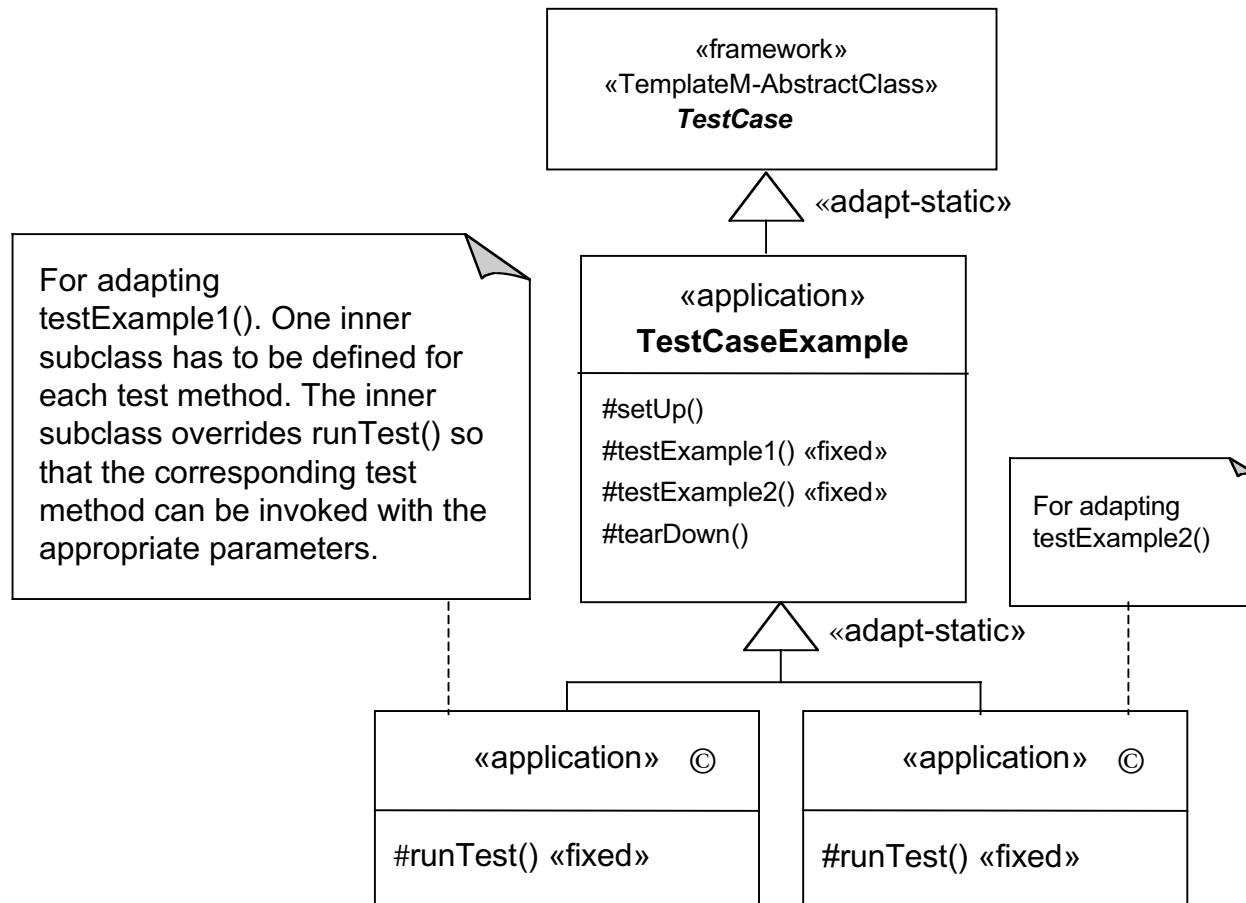
For possible adaptation examples, considering the optional hook methods



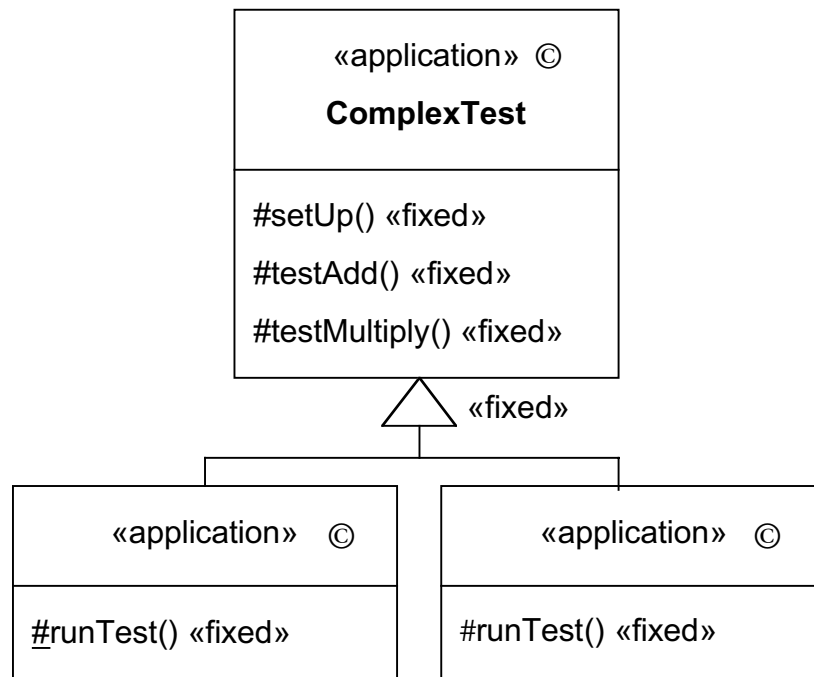
Adapting TestCase (IV)

- One aspect in the TestCase class cannot be captured in UML-F design diagrams
 - Method runTest() takes no parameters as input
 - Different test cases require different input parameters.
 - The interface for these test methods has to be adapted to match runTest().

Adapting TestCase (V)



Adapting TestCase (VI)



```
public class ComplexTest extends TestCase {
    private ComplexNumber fOneZero;
    private ComplexNumber fZeroOne;
    private ComplexNumber fMinusOneZero;
    private ComplexNumber fOneOne;
```

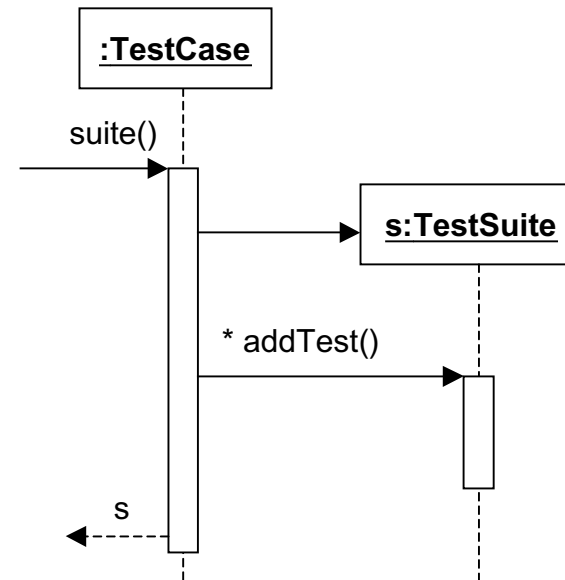
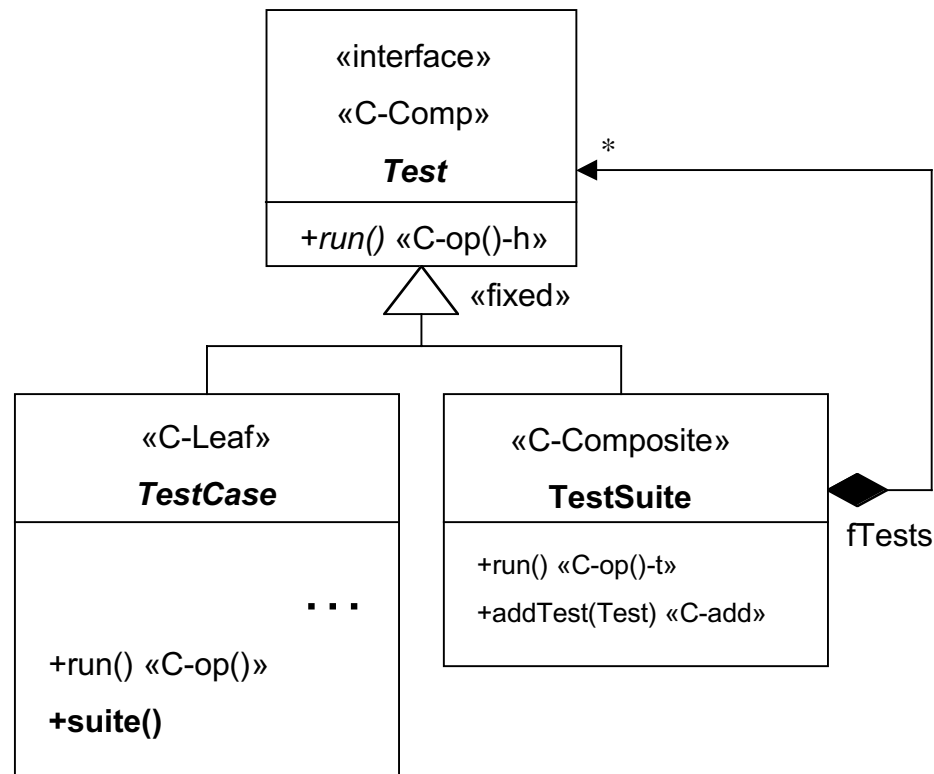
```
protected void setUp() {
    fOneZero = new ComplexNumber(1, 0);
    fZeroOne = new ComplexNumber(0, 1);
    fMinusOneZero = new ComplexNumber(-1, 0);
    fOneOne = new ComplexNumber(1, 1);
}
```

```
public void testAdd() {
    //This test will fail !!!
    ComplexNumber result = fOneOne.add(fZeroOne);
    assert(fOneOne.equals(result));
}
```

```
public void testMultiply() {
    ComplexNumber result = fZeroOne.multiply(fZeroOne);
    assert(fMinusOneZero.equals(result));
}
```

Adapting TestSuite (I)

Adaptation by overriding the **suite()** method

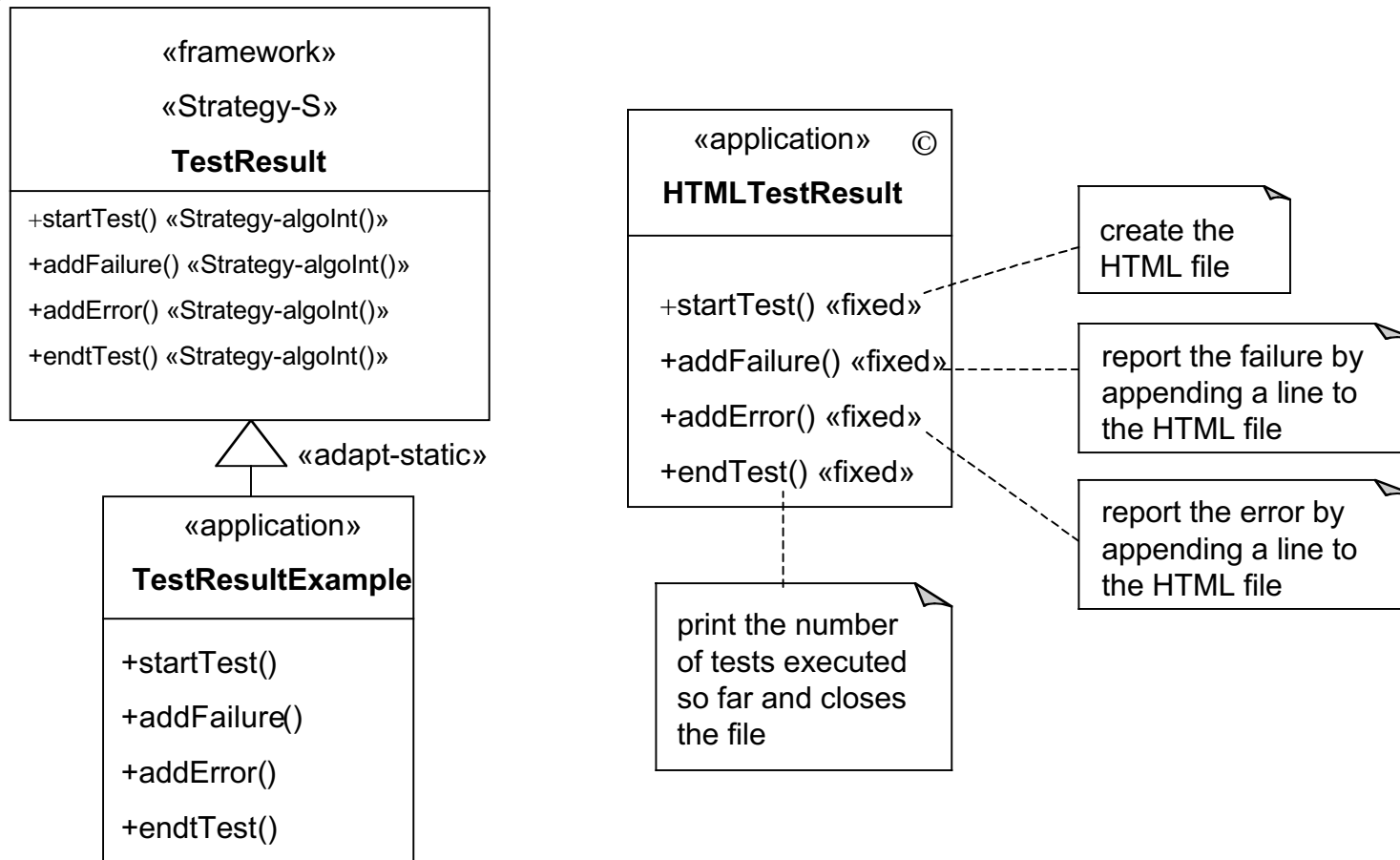


Adapting TestSuite (II)

TestCase and TestSuite are related variation points

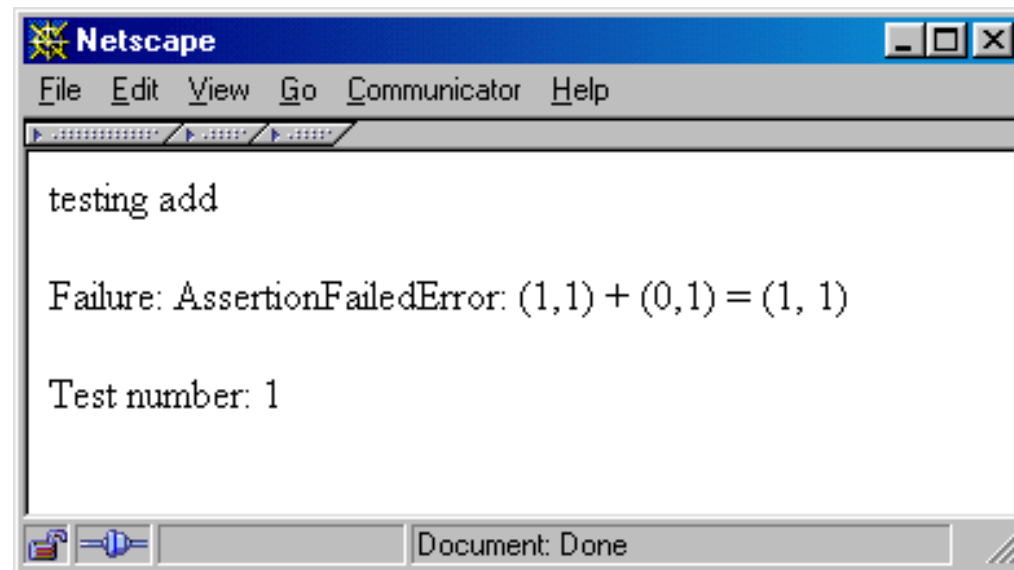
```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new ComplexTest("testing add") {
        protected void runTest() { this.testAdd(); }
    });
    suite.addTest(new ComplexTest("testing multiply") {
        protected void runTest() { this.testMultiply(); }
    });
    return suite;
}
```

Adapting TestResult (I)



Adapting TestResult (II)

Display of a sample HTML file that reports a failure.



Pattern-annotated diagrams

Pattern-annotated diagram for the main JUnit classes

