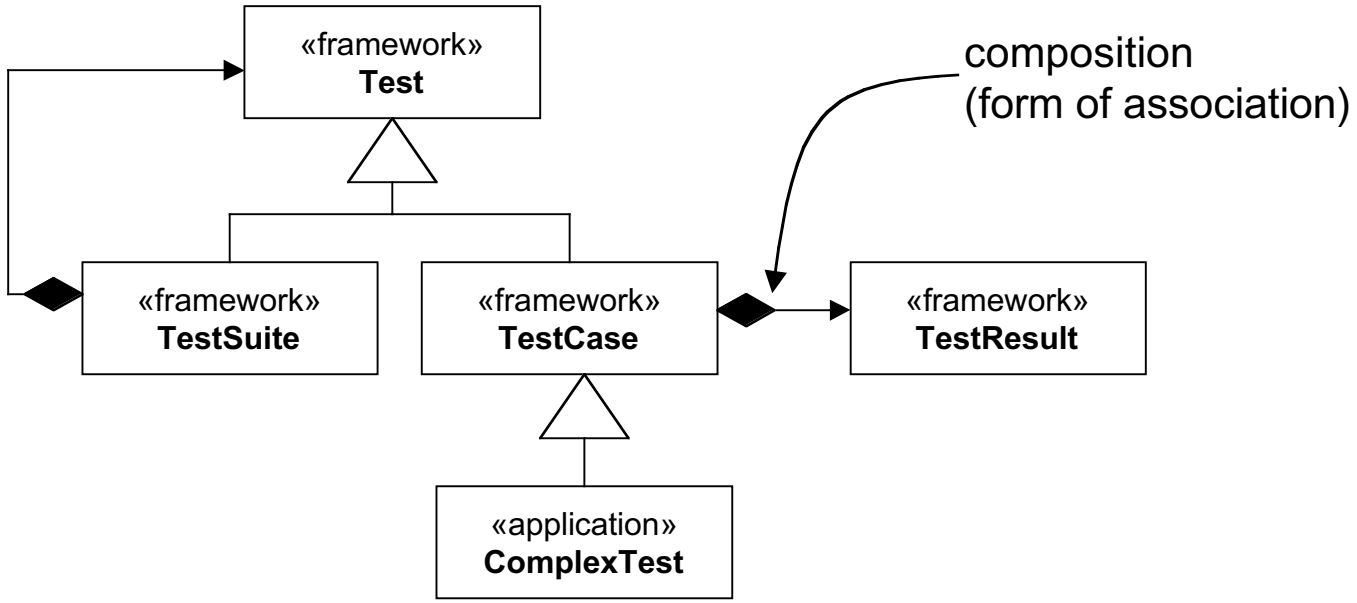
The JUnit 3.x testing framework

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

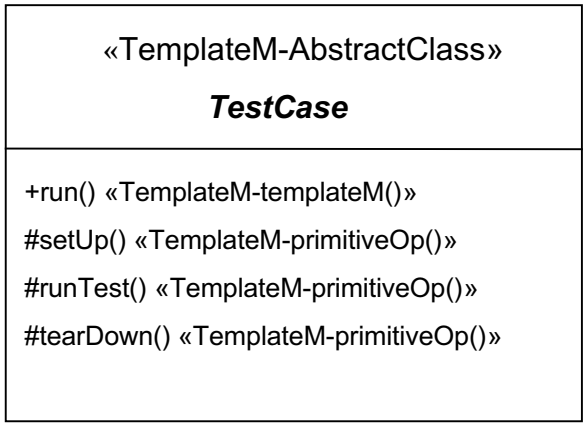


The TestCase variation point (I)

- 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)

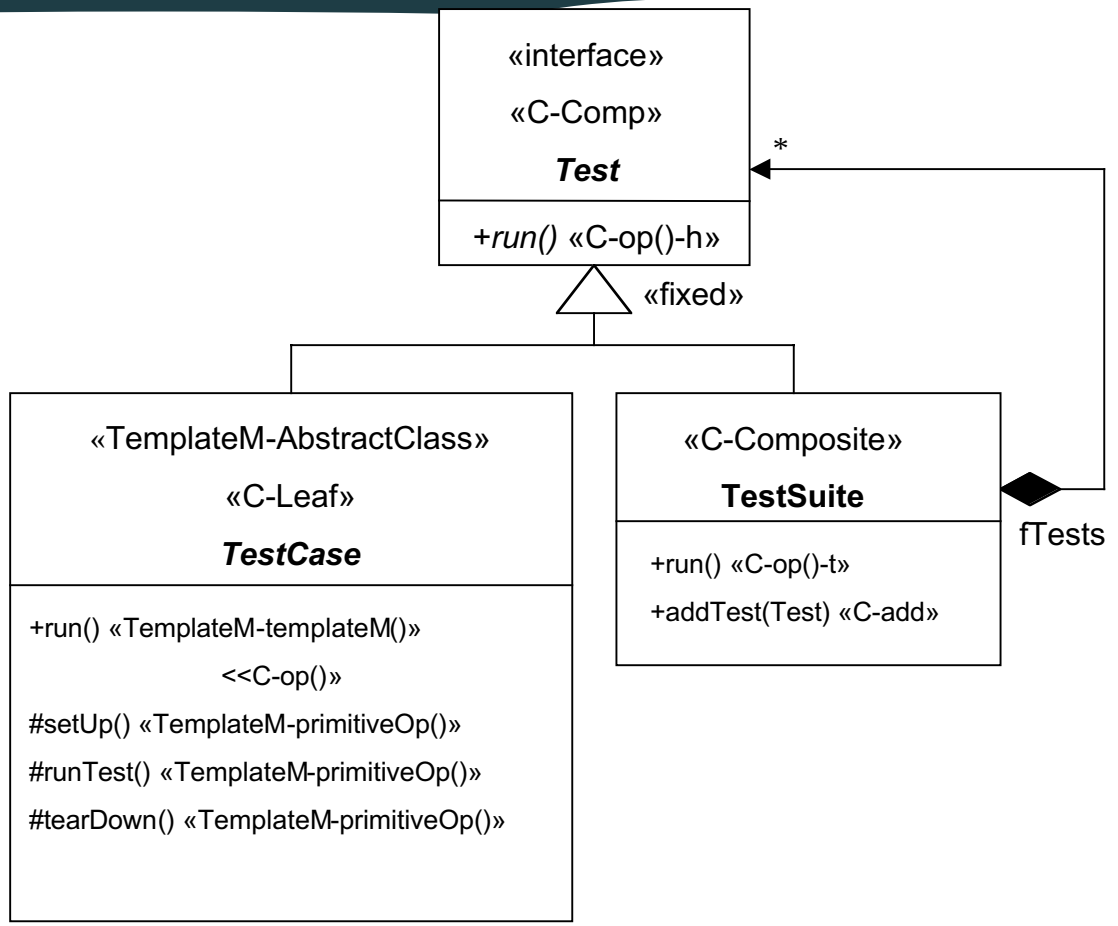
The TestCase design is based on the Template Method design pattern - method run() controls the test execution




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

The TestSuite variation point

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



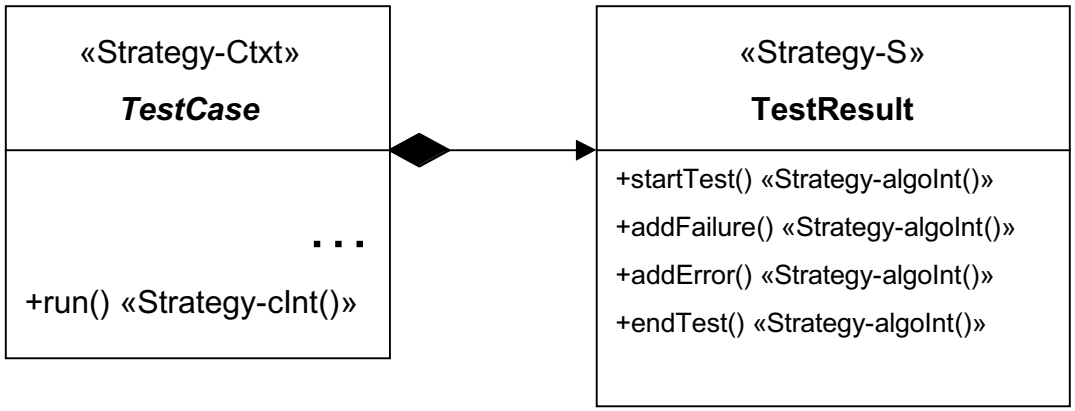
Black-box adaptation

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.

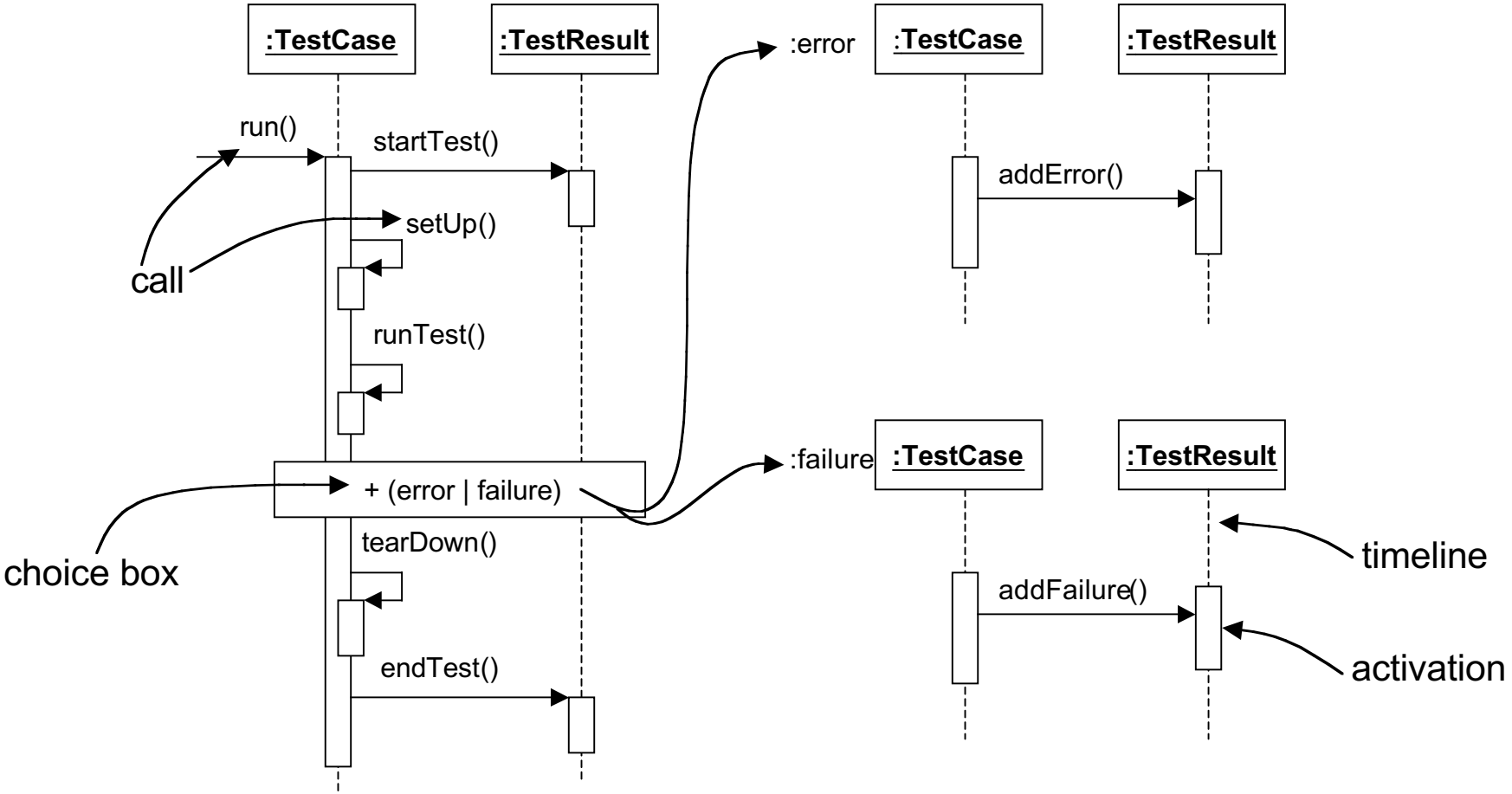
The TestResult variation point (II)

- 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

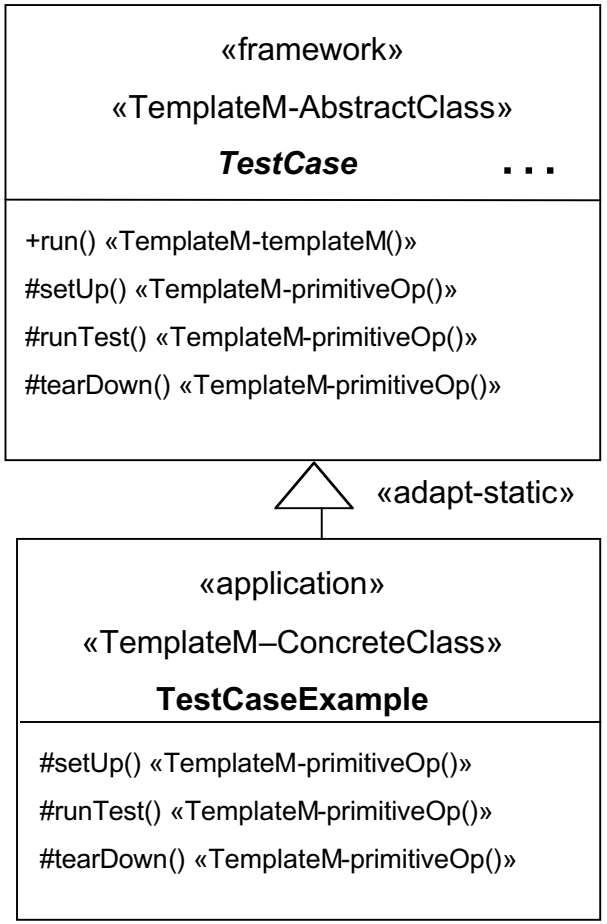
Adapting TestCase (I)

- 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



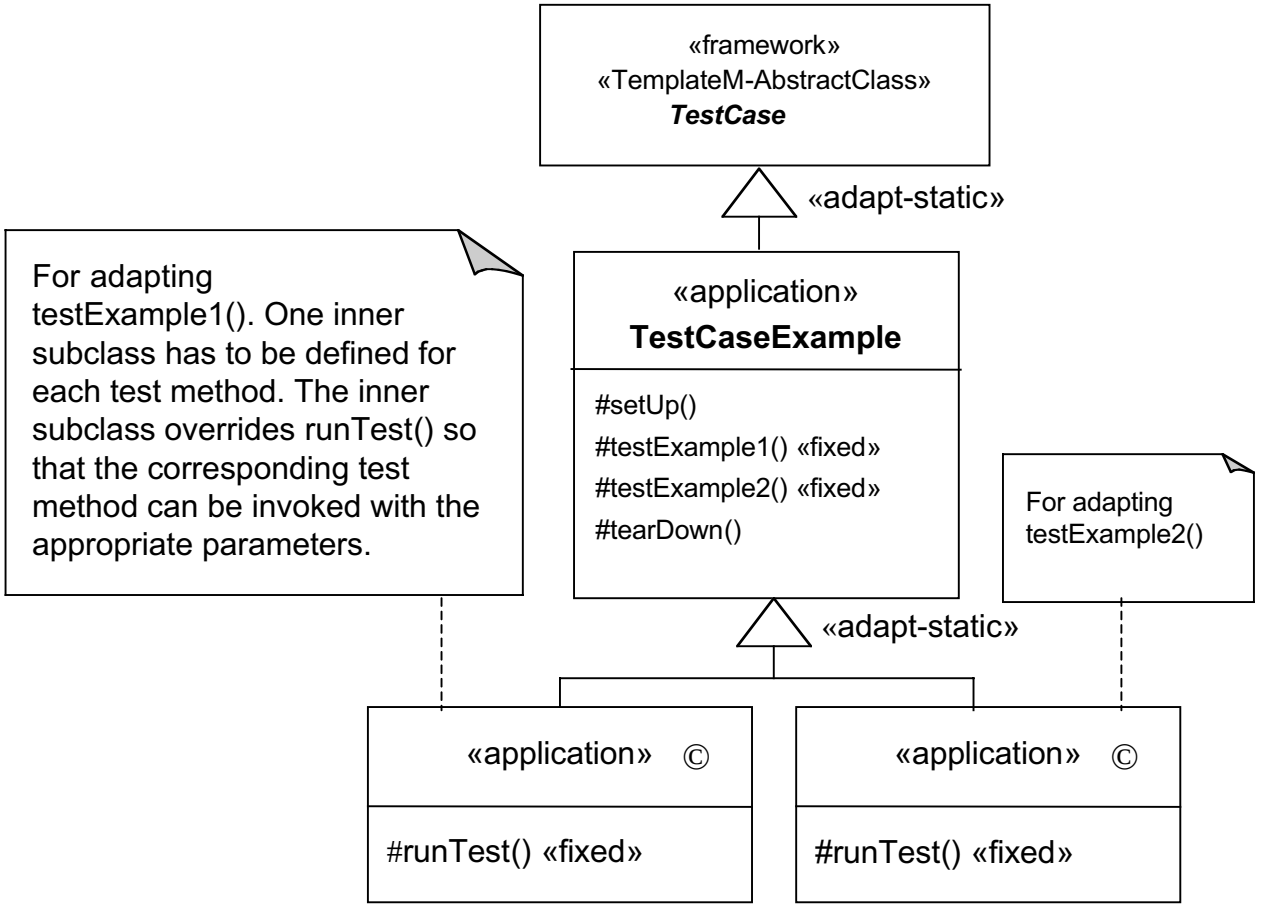
Adapting TestCase (III)

Four possible adaptation examples, considering the optional hook methods

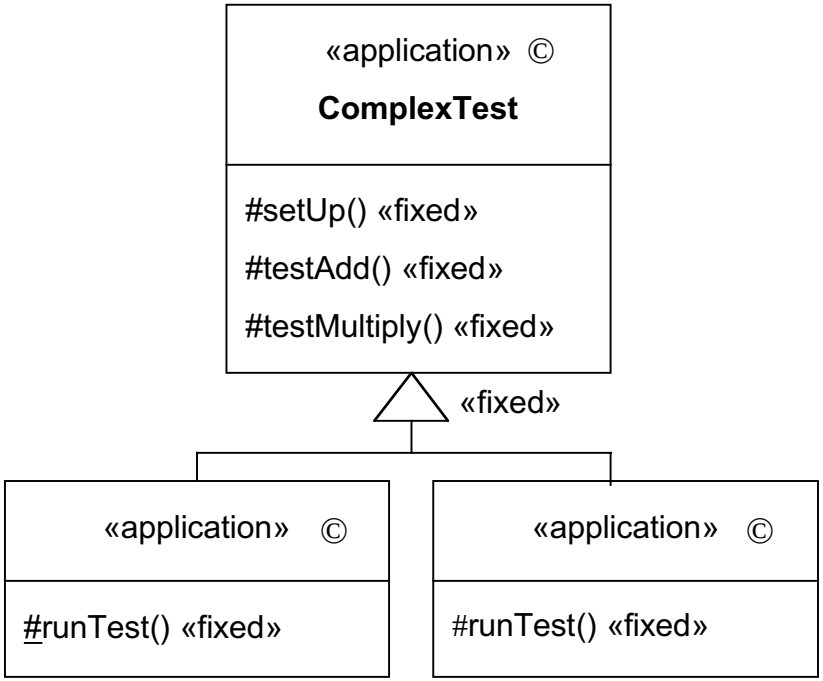
Adapting TestCase (IV)

- 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 theses test methods has to be adapted to match runTest().

Adapting TestCase (V)



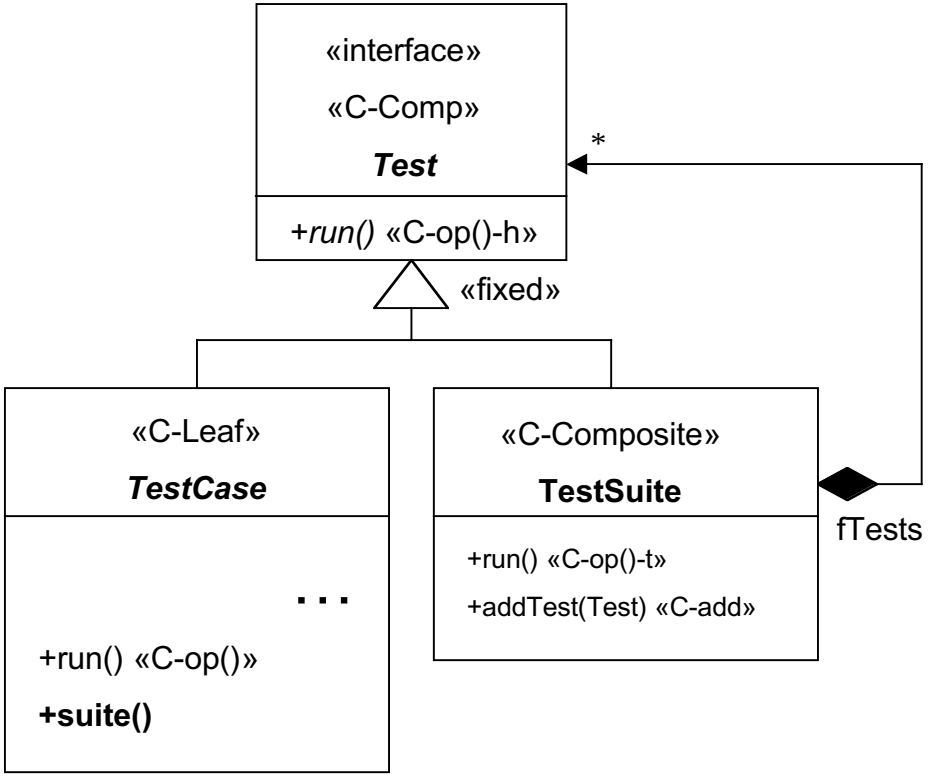
Adapting TestCase (VI)

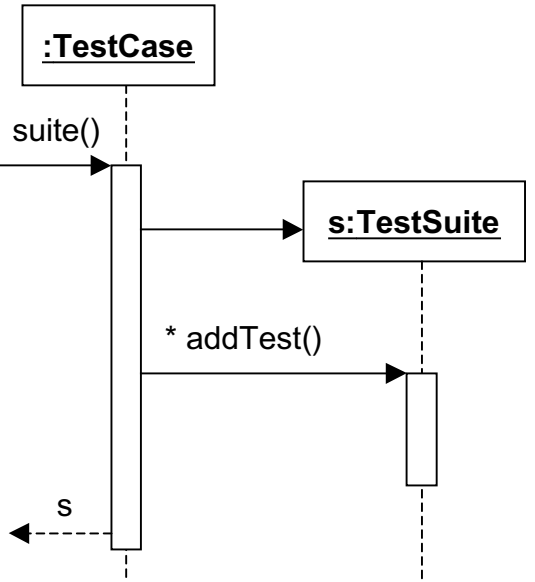



```
public class ComplexTest extends TestCase {
        private ComplexNumber fOneZero;
        private ComplexNumber fZeroOne;
        private ComplexNumber fMinusOneZero;
        private ComplexNumber fOneOne;

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



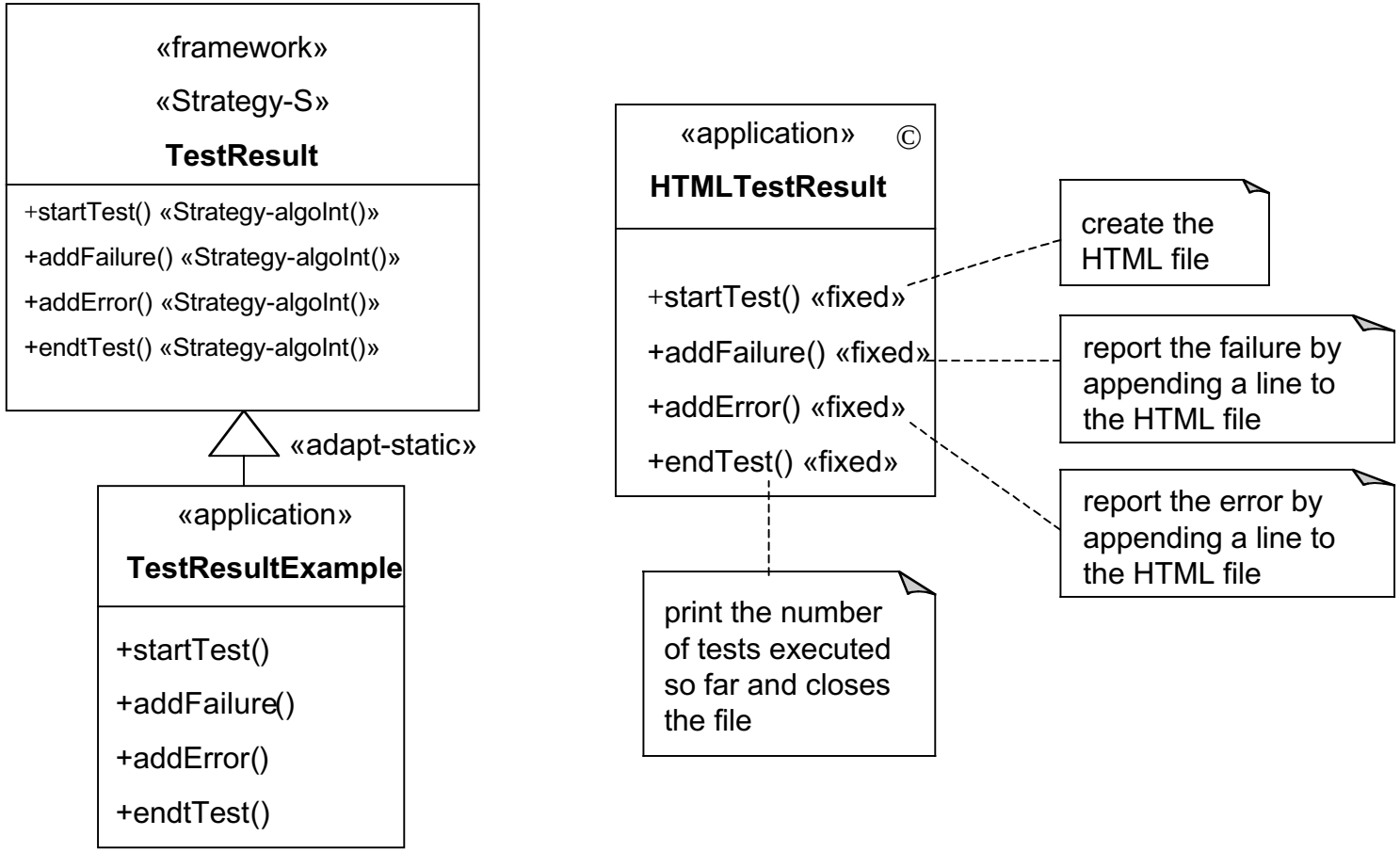


Adapting TestSuite (II)

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)

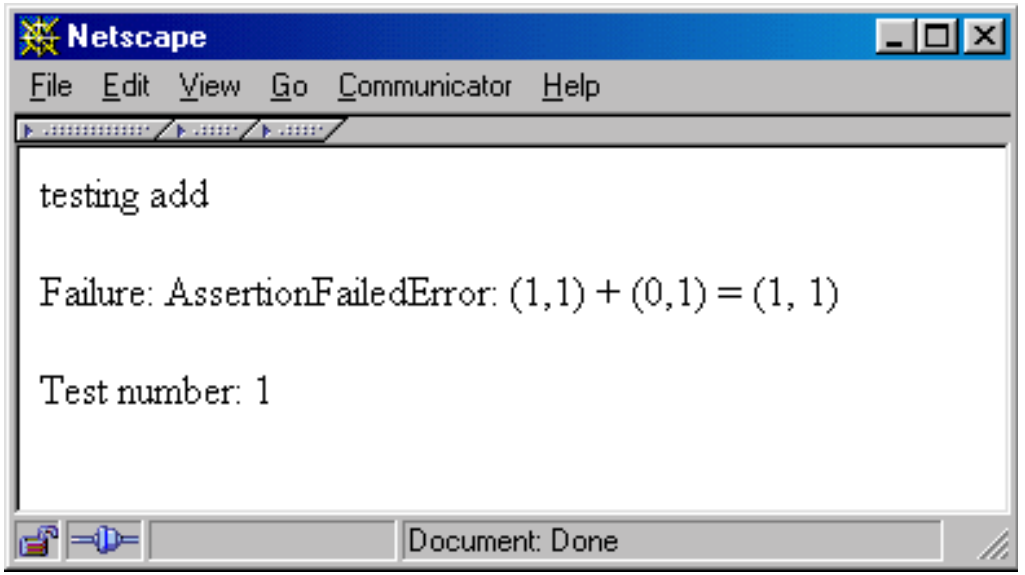


Adapting TestResult (II)

Display of a sample

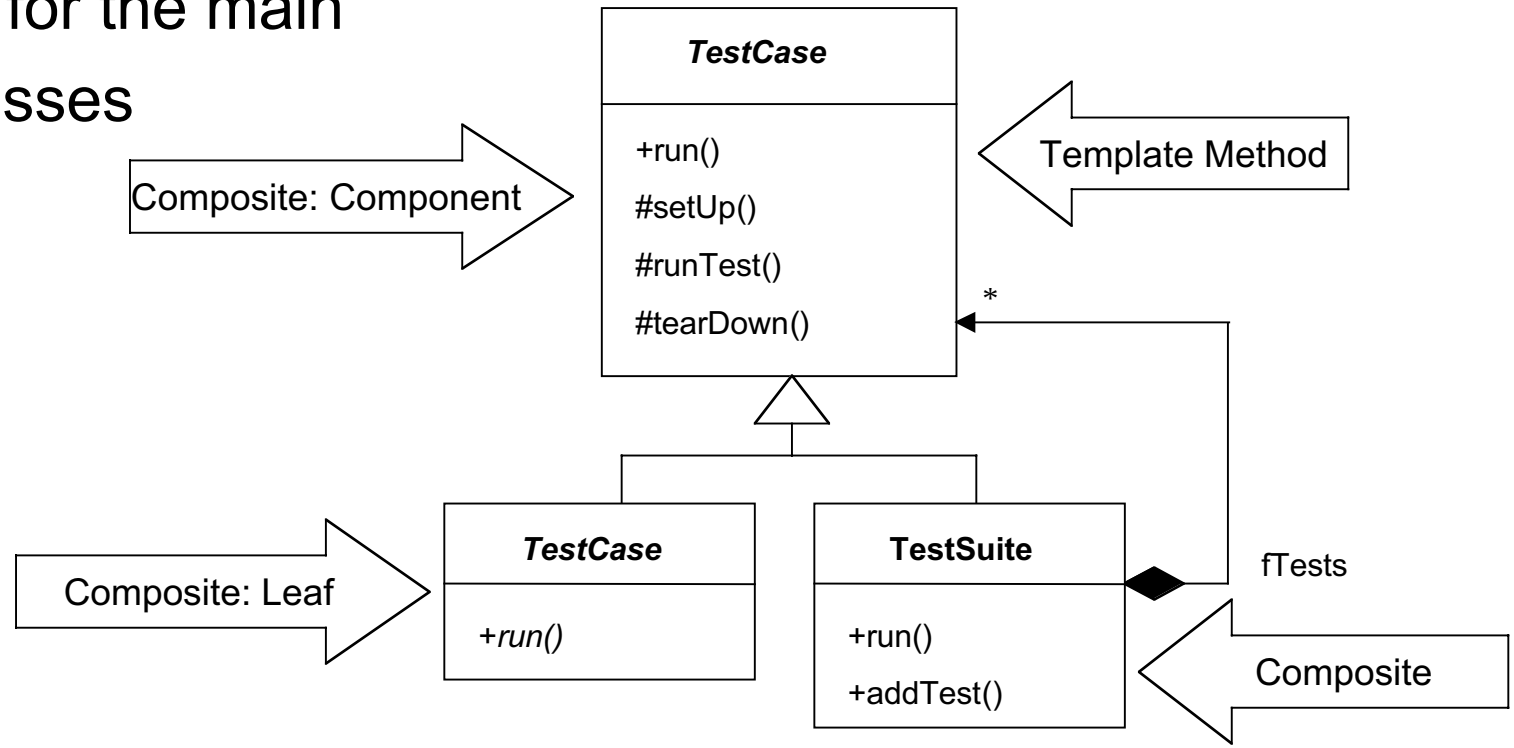
HTML file that reports

a failure.



Pattern-annotated diagrams

Pattern-annotated diagram for the main JUnit classes



Using reflection for adaptation

Example of using reflection:

```
Class c = this.getClass();
Method m[ ] = c.getDeclaredMethods();

for (int i = 0; i < m.length; i++){
        String methodName = m[i].getName();
        ...
        if (m[i].getParameterTypes().length == 0){
            m[i].invoke(this);
        }
}
```

From JUnit 3.x to JUnit 4.x

(an annotation-based framework)

UNIVERSITÄT SALZBURG

Simplified usage through **annotations**

- JUnit 4 requires Java 5 or newer
- No TestCase class
  - Use an ordinary class for a test case (don't extend TestCase)
- Use *annotations* instead of special method names:
  - Instead of a **setUp** method, put **@Before** before some method
  - Instead of a **tearDown** method, put **@After** before some method
  - You can define multiple test methods with any names, just put **@Test** before each test method
- Useful assertion methods are defined in the static-only class Assert

Java annotations in a nutshell

- From J2SE 5 up
- Allow adding decorations to code (resemble Java tags, e.g., @*author*)
- Used for code documentation, compiler processing, code generation, runtime processing
- Small set of predefined annotations (e.g., @*Deprecated*)
- New annotations can be created by developers

The ComplexTest Example in Junit 4.x

```
import org.junit.*;
import static org.junit.Assert.*;

public class ComplexTest {
    private ComplexNumber fZeroOne;
    private ComplexNumber fMinusOneZero;
    private ComplexNumber fOneOne;
    @Before
    public void setUp() {
        fZeroOne = new ComplexNumber(0, 1);
        fMinusOneZero = new ComplexNumber(-1, 0);
        fOneOne = new ComplexNumber(1, 1);
    }
    @Test
    public void testAdd() {          //This test will fail !!!
        ComplexNumber result = fOneOne.add(fZeroOne);
        assertTrue(fOneOne.equals(result));
    }
    @Test
    public void testMultiply() {
        ComplexNumber result = fZeroOne.multiply(fZeroOne)
        assertTrue(fMinusOneZero.equals(result));
    }
}
```

UNIVERSITÄT SALZBURG

Before and After

- *@BeforeClass*: one-time initialization, when the class is loaded
- *@AfterClass*: clean-up method, after all tests have been completed
- Multiple *@Before* and *@After* methods possible, no order assumed – for example:

```
@Before
public void setUpZeroOne() { fZeroOne = new ComplexNumber(0, 1); }

@Before
public void setUpMinusOneZero() {fMinusOneZero = new ComplexNumber(-1, 0);}

@Before
public void setUpOneOne() { fOneOne = new ComplexNumber(1, 1); }
```

UNIVERSITÄT SALZBURG

Additional features

- Execution time limit to avoid infinite loops
  - *@Test (timeout = 10)*
- Specification of expected exceptions
  - *@Test (expected = ArrayIndexOutOfBoundsException.class)*
  -
- Parameterized tests: A series of tests which differ only in the inputs and expected results can be written as a single test by providing:
  - A static method that generates and returns a list of data items
  - A single constructor that stores one data item to test
  - A test method

Example of parameterized test

```
@RunWith(Parameterized.class)
public class ComplexTestParam {
    private ComplexNumber fZeroOne;
    private ComplexNumber expectedResult;
    private ComplexNumber secondTerm;

    public ComplexTestParam(ComplexNumber expected, ComplexNumber secondTerm) {
          this.expectedResult = expected;
          this.secondTerm = secondTerm;
          fZeroOne = new ComplexNumber(0, 1);
    }
    @Parameters
    public static Collection<Object[]> generateInputs() {
          return Arrays.asList(new Object[][]{
          {new ComplexNumber(-1,0), new ComplexNumber(0,1)},
          {new ComplexNumber(1,0), new ComplexNumber(1,0)}  //wrong
          });
        }
    @Test
    public void testMultiply() {
        ComplexNumber result = fZeroOne.multiply(secondTerm)
        assertTrue(result.equals(expectedResult));
    }
}
```

UNIVERSITÄT SALZBURG

Test suites in Junit 4.x

- A test suite is defined by an annotation to some (possibly empty) class

- Example:

```
@RunWith(Suite.class)
@Suite.SuiteClasses( { ComplexTest.class, ComplexTestParam.class })
public class AllComplexTests {
//empty
}

// Complete source code available online (see course homepage)
```

UNIVERSITÄT SALZBURG

Actor-Oriented Design and The Ptolemy II framework

http://ptolemy.eecs.berkeley.edu/

UNIVERSITÄT SALZBURG

Ptolemy II objectives

- Supports modeling, simulation and design of concurrent systems
- Promotes component-based modeling, where a component represents a domain-specific entity and it is called an **actor**
- Provides widely-used models of interaction between components, called **models of computation**
- Advocates a programming discipline called **Actor-Oriented Programming**
- Focuses on flexibility
- Encourages experimentation with designs

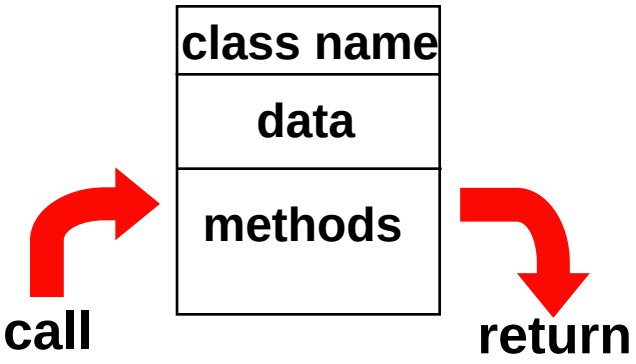
Actor Oriented Design

- Actors are conceptually concurrent (no predefined order of execution)
- Actors interact by sending messages through channels
- An actor implements an execution interface
- Dedicated components are responsible for data transfer between actors and for executing the actors
- Actors can be hierarchically composed

UNIVERSITÄT SALZBURG

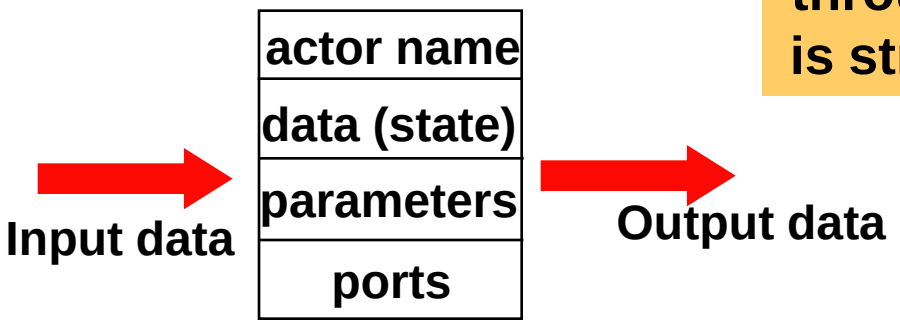
AOD versus OOD* (I)

Object orientation:



What flows through an object is sequential control

Actor orientation:



What flows through an object is streams of data

UNIVERSITÄT SALZBURG

* Edward Lee, UC Berkeley

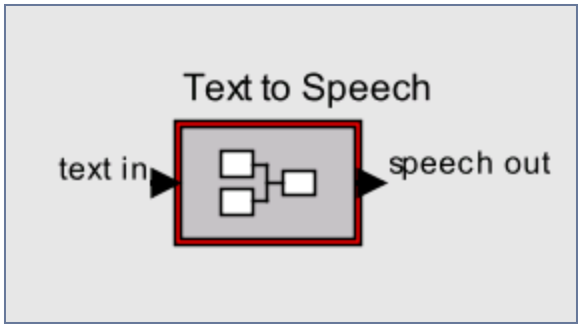
AOD versus OOD* (II)

Object oriented

TextToSpeech
initialize(): void notify(): void isReady(): boolean getSpeech(): double**[]**

OO interface definition gives procedures that have to be invoked in an order not specified as part of the interface definition.

Actor oriented



actor-oriented interface definition says "Give me text and I'll give you speech"

- Identified limitations of object orientation:
  - Says little or nothing about concurrency and time
  - Concurrency typically expressed with threads, monitors, semaphores
  - Components tend to implement low-level communication protocols
    - Re-use potential is disappointing

UNIVERSITÄT SALZBURG

* Edward Lee, UC Berkeley

Examples of Actor-Oriented component frameworks*

- Simulink (The MathWorks)
- Labview (National Instruments)
- Modelica (Linkoping)
- OPNET (Opnet Technologies)
- Polis & Metropolis (UC Berkeley)
- Gabriel, Ptolemy, and Ptolemy II (UC Berkeley)
- OCP, open control platform (Boeing)
- GME, actor-oriented meta-modeling (Vanderbilt)
- SPW, signal processing worksystem (Cadence)
- System studio (Synopsys)
- ROOM, real-time object-oriented modeling (Rational)
- Easy5 (Boeing)
- Port-based objects (U of Maryland)
- I/O automata (MIT)
- VHDL, Verilog, SystemC (Various)
- …

UNIVERSITÄT SALZBURG

* Edward Lee, UC Berkeley

Live examples of actor-oriented designs

- Matlab/Simulink: Automotive controllers
- Ptolemy: Finite State Machine example