

# Orga Item

- ◆ Time Monitoring during: Mid-semester break?
  - → Yes!
- ◆ Work expected during break?
  - → No!

# **CSSE2003**

## **Software Engineering Studio**

Semester 2, 2009

### **15: A Design Example: JUnit**

# Lecture Summary

- ◆ Design of JUnit:
  - A key requirement
  - Design of the TestCase class
    - ◆ Using the Command pattern
    - ◆ Using the Template pattern
  - Handling failures and errors
    - ◆ Using exceptions
  - More on the TestCase design
    - ◆ Using the Adapter pattern
  - Design of the TestSuite class
    - ◆ Using the Composite pattern

# Key Requirement: Testing Made Easy

- ◆ “number one goal is to write a framework within which ... developers will actually write tests. It has to use familiar tools, so there is little new to learn. It has to require no more work than absolutely necessary to write a test. It has to eliminate duplicated effort.” - JUnit A Cook’s Tour (JACT)
- ◆ JUnit defines tests as objects because
  - objects can be manipulated
  - objects can be reused
  - objects are familiar to OO programmers

# The TestCase Class

- ◆ “second goal ... creating tests that retain their value over time” (JACT) - make a test an object

```
class TestCase {  
    String fName;  
    public TestCase (String name) {  
        fName = name;  
    }  
}
```

- ◆ Attribute fName gives a test object a name

```
TestCase test =  
    new FaultTest("testFault1");  
test.run();
```

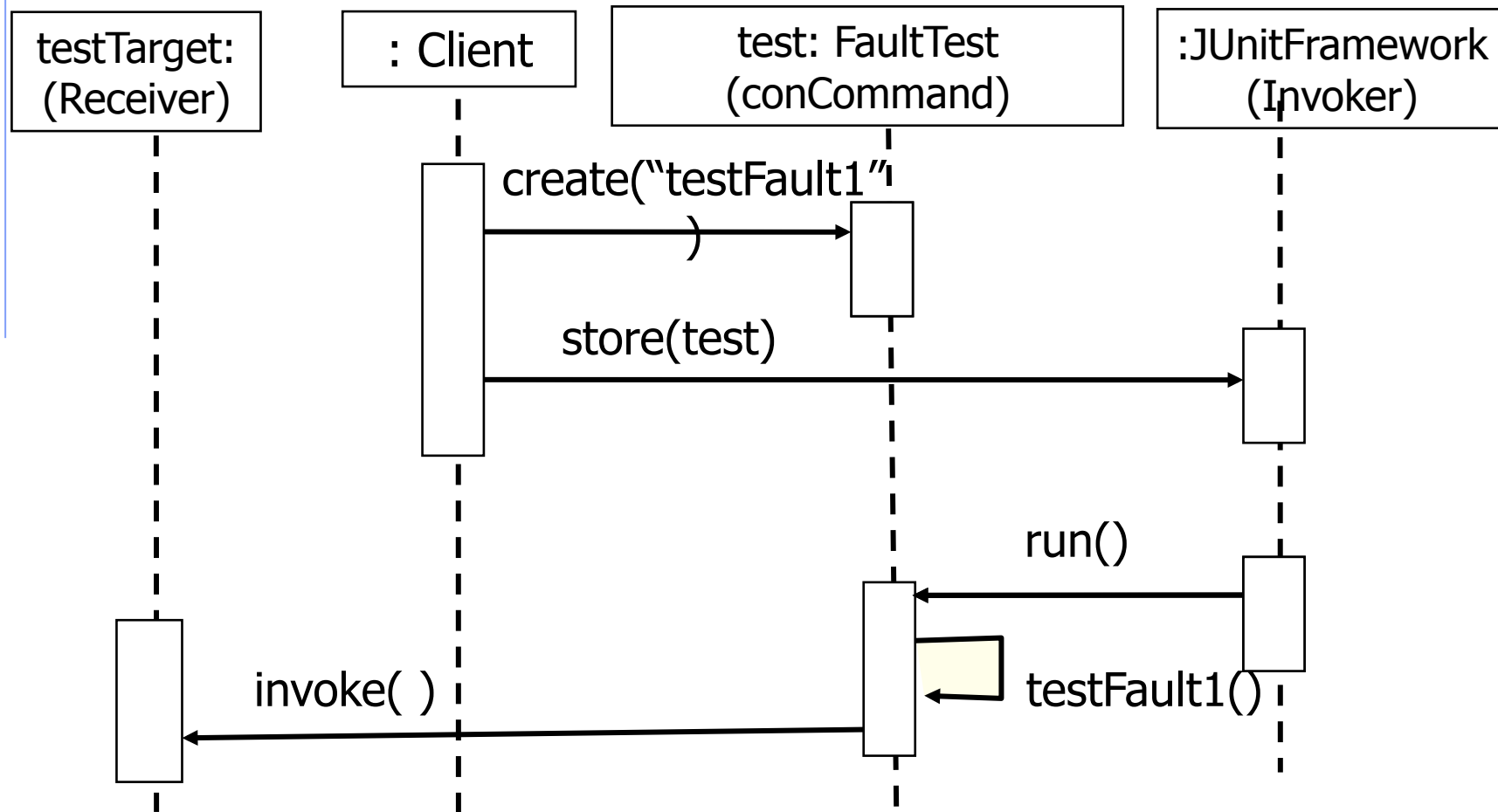
## The Testcase Class: The Command Pattern

- ◆ Behavioural pattern Command: capture an operation as an object [Larman, p.643]
- ◆ A test embodies an operation, represented by method `run()`

```
class TestCase {  
    run () {...}  
}
```



# Command Pattern Use in JUnit

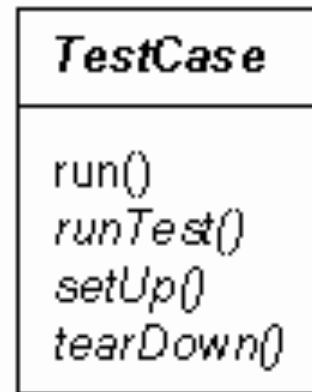


## The Testcase Class: The Template Pattern

- ◆ “It should be possible to combine tests ... without fear of interference. ... Fortunately, there is a **common structure to all tests** – they set up a test fixture, run some code against the fixture, check some results, and then clean up the fixture.” (JACT).
- ◆ Behavioural pattern Template: allows runtime variants of an algorithm, express the algorithm in a base class [Larman, p.630]
- ◆ The Template pattern is about defining the shape of an algorithm in a method, but deferring details to a subclass. The subclass defines the details without changing the algorithm structure

# The Template Pattern - Testcase

```
abstract class TestCase {  
    run() {  
        setUp();  
        runTest();  
        tearDown();  
    }  
}
```

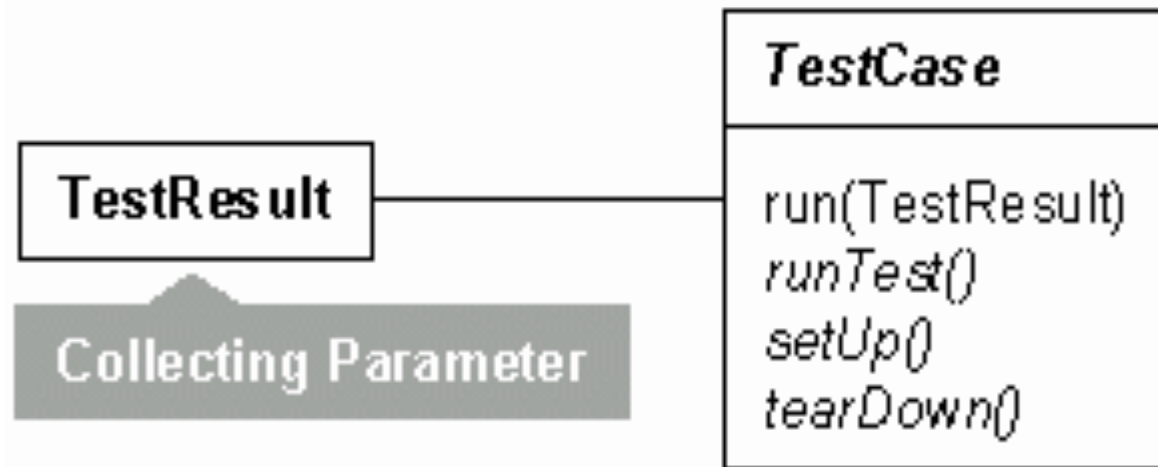


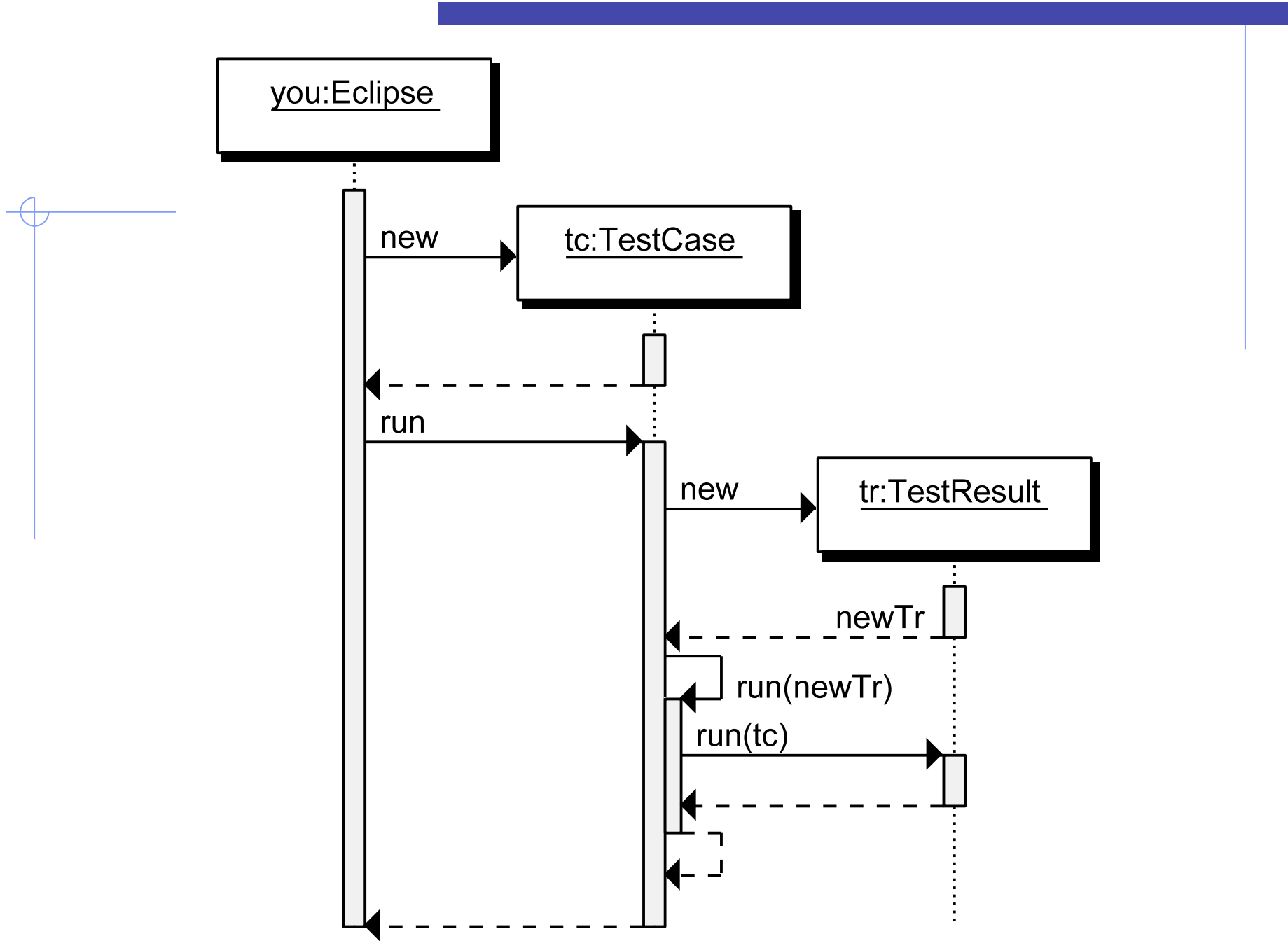
Template Method

- ◆ TestCase is made abstract because actual behaviour of `run()` will depend on a subclass
- ◆ Default `setUp()` and `tearDown()` do nothing

# The Testresult Class

- ◆ Ease of use requires a simple way to record test results
- ◆ “when you need to collect results over **several methods**, add a parameter to the method and pass an object that will collect the results” (JACT)





## In TestCase Class

```
abstract class TestCase {
    public TestResult run() {
        TestResult result= createResult();
        run(result);
        return result;
    }
    public void run(TestResult result) {
        result.run(this);
    }
}
```

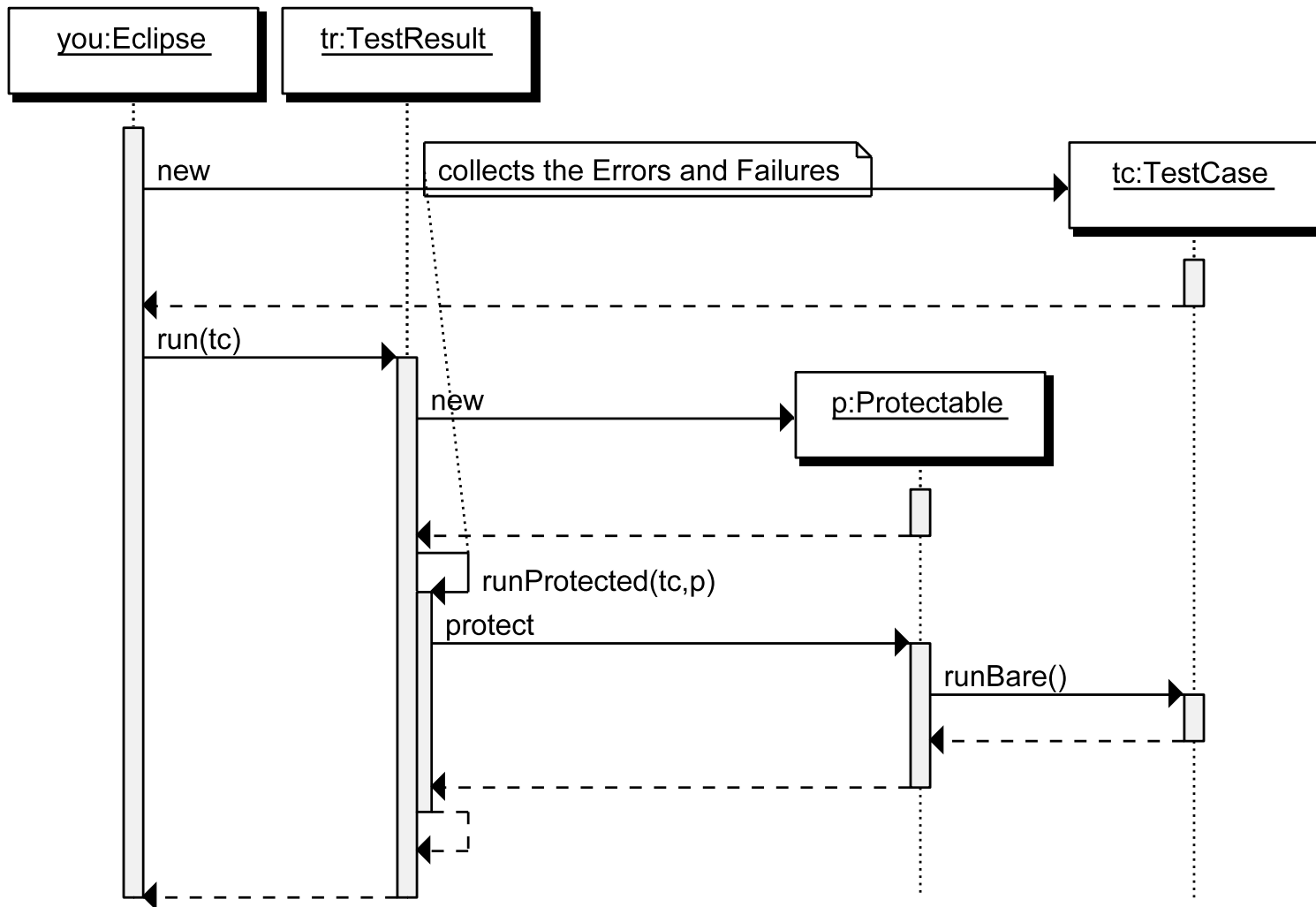
# Failures And Errors

Asymmetry: **failures** vs. **errors**

- ◆ **Failure** : Anticipated & checked for with assertions  
→ Abort the Test
- ◆ **Error** : Unanticipated (e.g. RuntimeExceptions)  
→ Continue the Test
- ◆ 'java.lang.Throwable':
  - Error → Don't catch this. Real stuff-up!
  - Exception → Ok to catch. Try a fix.

**!! Name Clash → 'Error' !!**

# How is it done?



## in `TestResult.runProtected()`:

```
try {  
    ... an Assertion ...  
}  
catch (AssertionFailedError e) {  
    addFailure(test, e);  
} ...  
catch (Throwable e) {  
    addError(test, e);  
} ...
```

# Protected Execution

```
public class TestResult extends Object {
    protected void run(final TestCase test) {

        Protectable p= new Protectable() {
            public void protect() throws Throwable
            {

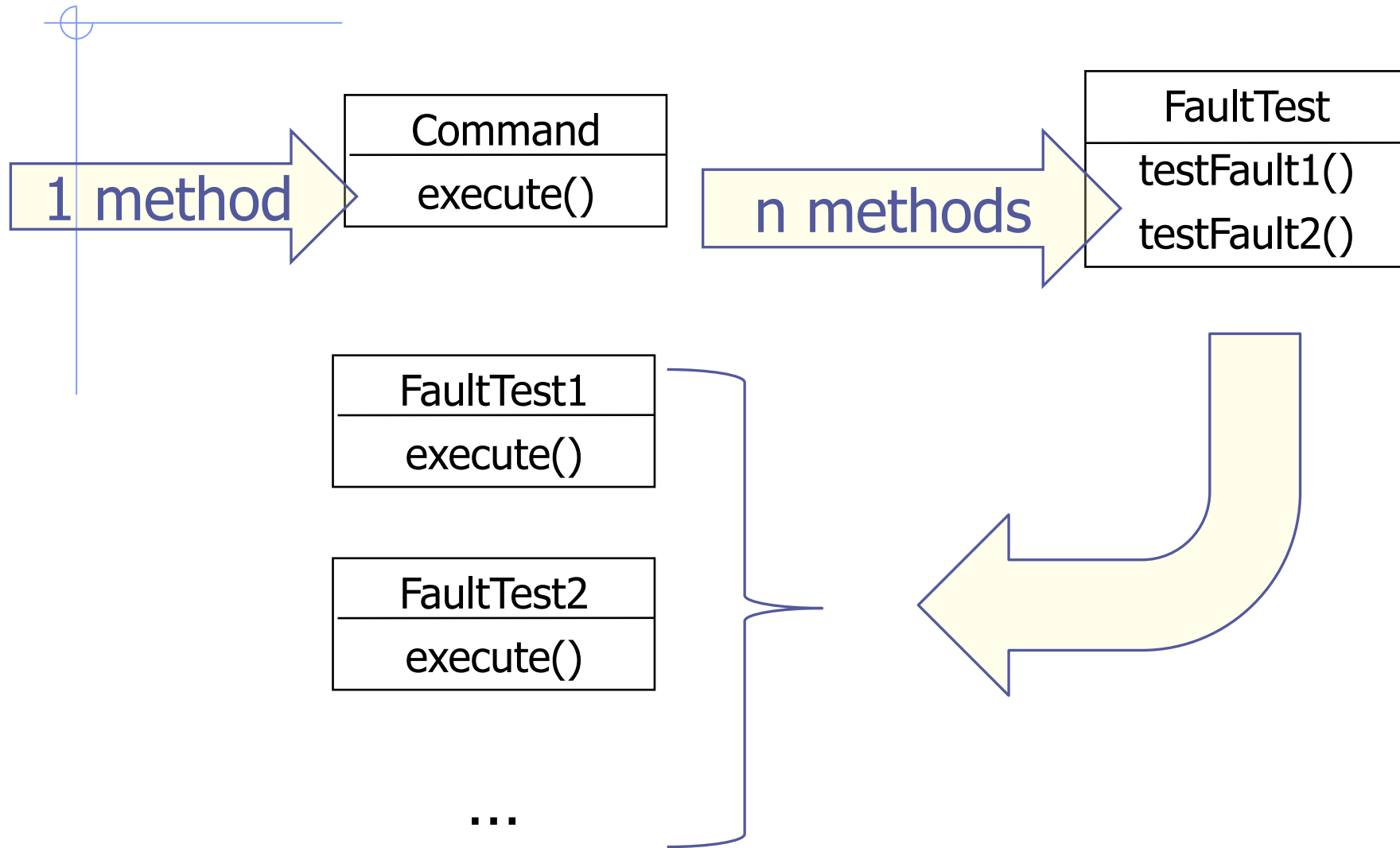
                test.runBare();

            }
        };
        runProtected(test, p);
    }
}
```

# The **Assertions** produce **Errors**

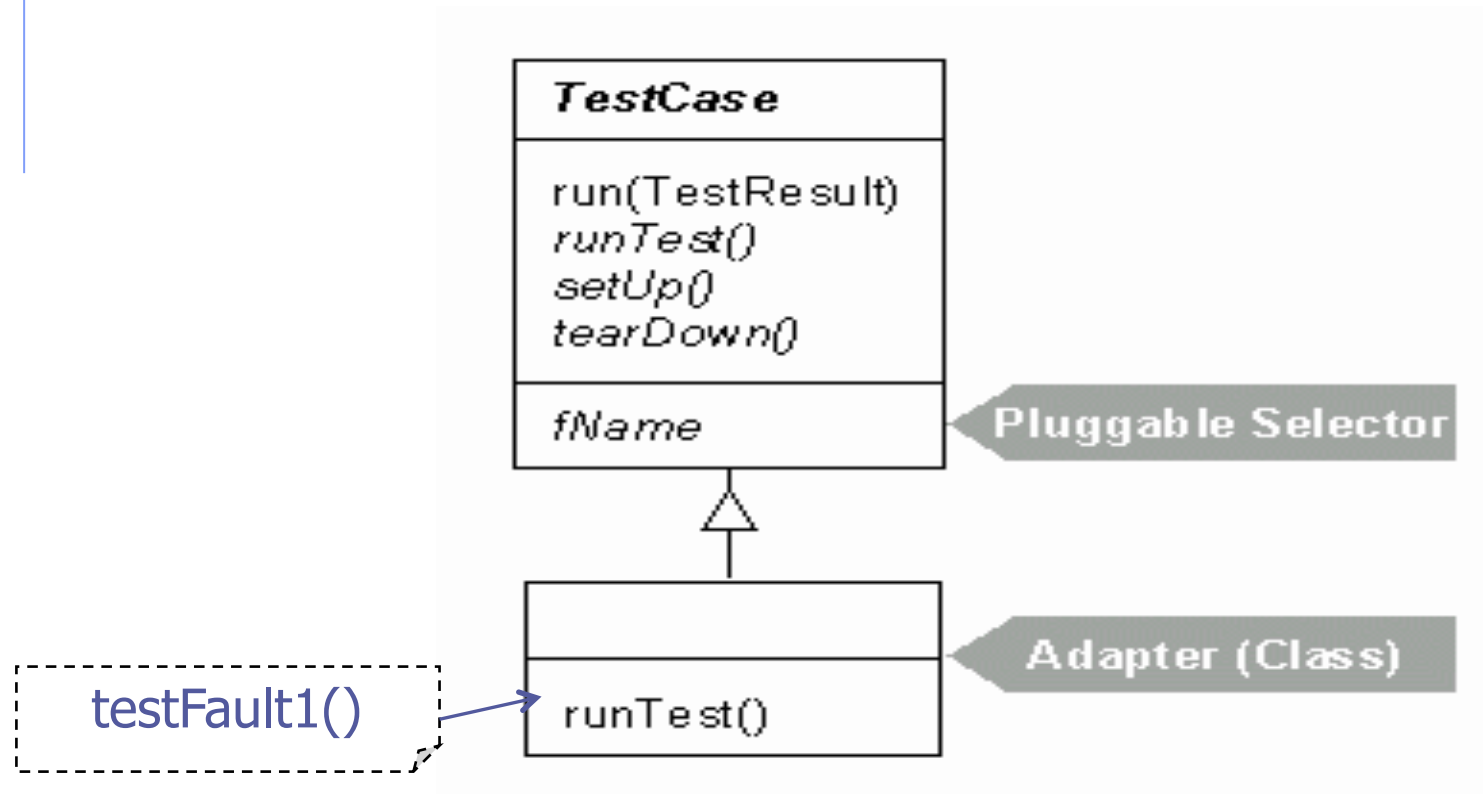
```
class Assert {
    assertTrue(message, condition) {
        if (! condition) fail(message);
    }
    fail(message) {
        throw new
AssertionFailedError (message) ;
    }
}
```

# Pattern Mismatch: Command



## Adapting Testcase to its Subclass -2

- ◆ *Adapter* pattern: converts the **actual** interface of a class (viz. test<MyTest>() ) into an interface that a client **expects** (viz. runTest() )



## (Clunky) Solution 1

- ◆ Override `runTest()` in a subclass of a subclass of `TestCase`

```
class TestFault1 extends FaultTest {  
    TestFault1() {super("testFault1");}  
    runTest() {testFault1();}  
}
```

...

```
TestCase test = new TestFault1();
```

- ◆ “subclassing requires us to implement **a subclass for each test [method]**. This is against the JUnit goal to make it simple to add a test case”

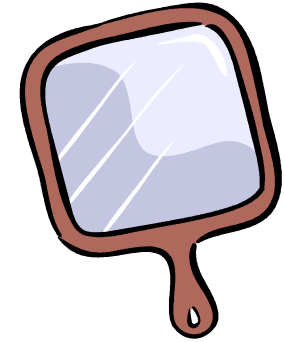
## (Clunky) Solution 2

- ◆ Override runTest() by defining an anonymous inner classes of the TestCase subclass

```
TestCase test =
    new FaultTest("testFault1")
    {
        runTest() { testFault1(); }
    };
```

- ◆ Tester still needs to define runTest()

## Solution 3: Use Reflection!

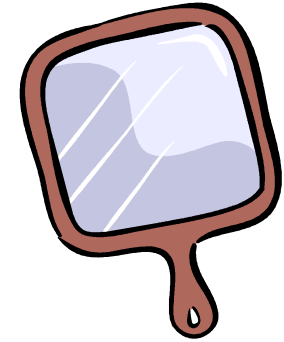


- ◆ Method `runTest()` defined in `TestCase` as

```
runTest() {  
    getClass().getMethod(fName, new  
        Class[0]).invoke(this, new  
        Class[0]);  
}
```

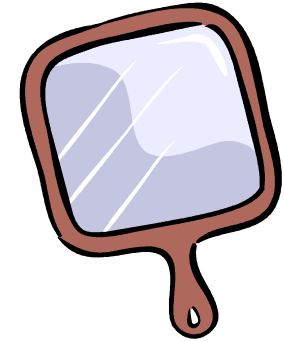
**Use WHAT?**

# What is Reflection?



- ◆ Reflection:  
An API for a running program to look at itself.
- ◆ `java.lang.reflect.Method`:  
An object that describes a Method!
- ◆ `invoke()`:  
A method that ...  
invokes an Object that ...  
describes a Method!

## Solution 3: Use Reflection



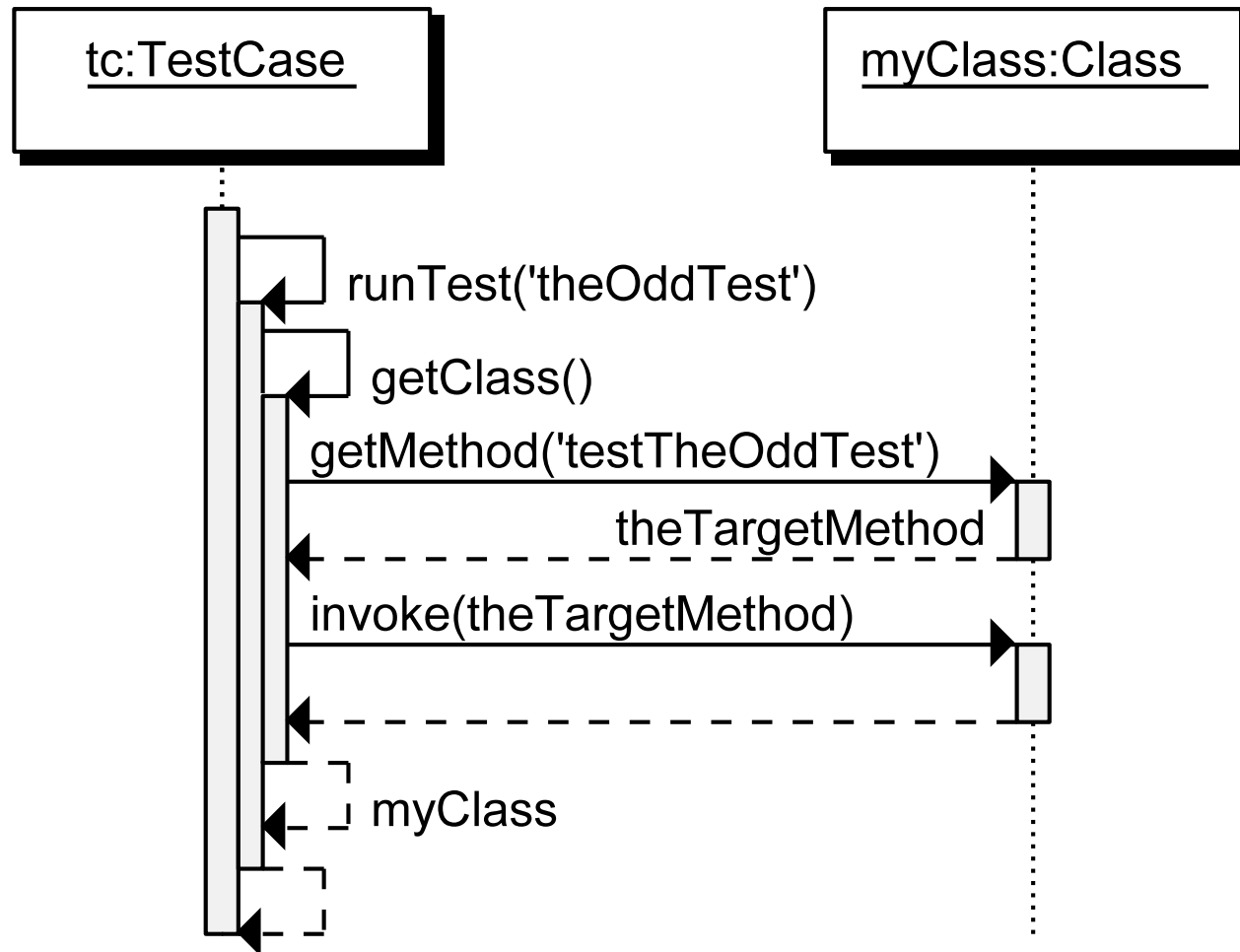
- ◆ Method `runTest()` defined in `TestCase` as

```
runTest() {  
    getClass().getMethod(fName, new  
    Class[0]).invoke(this, new  
    Class[0]);  
}
```

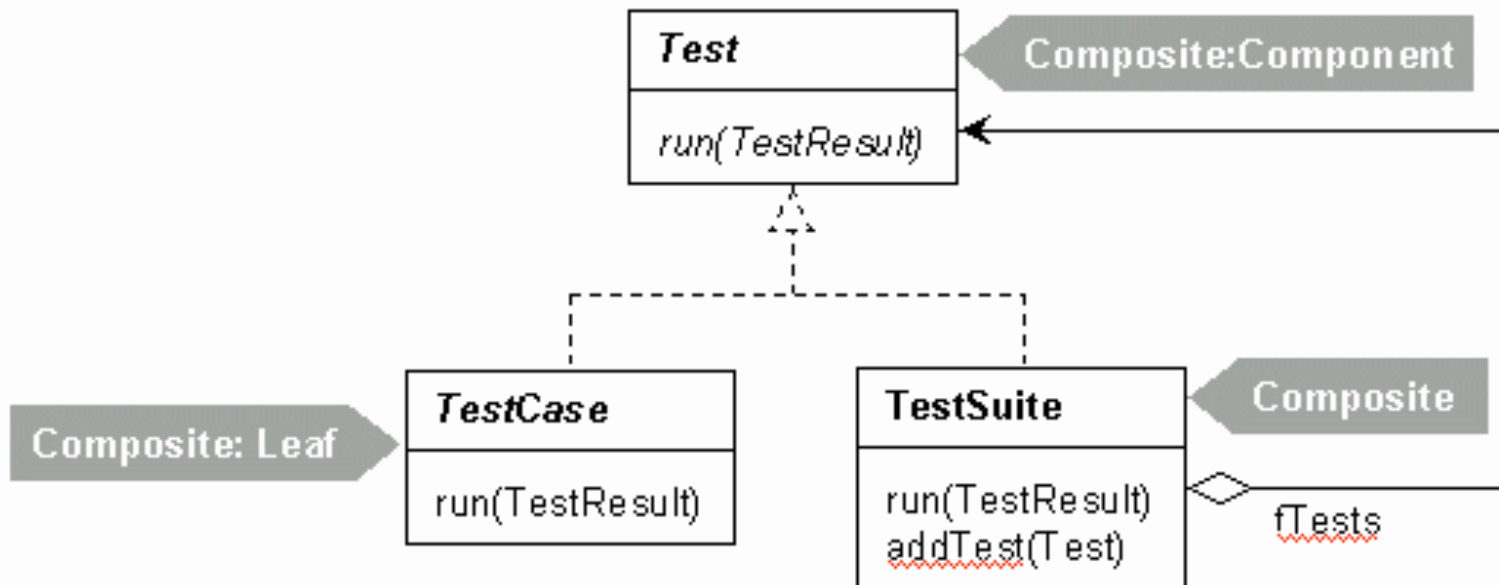
TestFault (Class)

testFault1 (Method)

# Scenario



# TestSuite: *Composing* Tests



## Collecting Results over a Suite

```
class TestSuite implements Test {
    Vector fTests = new Vector();
    run(TestResult result) {
        Enumeration e =
        fTests.elements();
        while (e.hasMoreElements()) {

            ( (Test)e.nextElement() ).run(result
            );
        } } }
```