b
UNIVERSITÄT
BERN

# 4. A Testing Framework

Oscar Nierstrasz

# A Testing Framework

## *Sources*

- JUnit documentation (from www.junit.org)

# Roadmap

- Junit — a testing framework
  - —Testing practices
  - —Frameworks vs. Libraries
  - —Junit 3.x vs. Junit 4.x (annotations)
- Money and MoneyBag — a testing case study
- Double Dispatch — how to add different types of objects
- JExample

# Roadmap

- **Junit — a testing framework**
  - —Testing practices
  - —Frameworks vs. Libraries
  - —Junit 3.x vs. Junit 4.x (annotations)
- Money and MoneyBag — a testing case study
- Double Dispatch — how to add different types of objects
- JExample

# The Problem

> *"Testing is not closely integrated with development. This prevents you from measuring the progress of development — you can't tell when something starts working or when something stops working."*
>
> *— Test Infected*

Interactive testing is tedious and seldom exhaustive.

*Automated tests* are better, but,

- —how to introduce tests interactively?
- —how to organize suites of tests?

# Testing Practices

***During Development***

- When you need to add new functionality, *write the tests first.*
  - You will be done when the test runs.
- When you need to redesign your software to add new features, refactor in small steps, and *run the (regression) tests after each step.*
  - Fix what's broken before proceeding.

***During Debugging***

- When someone discovers a defect in your code, *first write a test* that demonstrates the defect.
  - Then debug until the test succeeds.

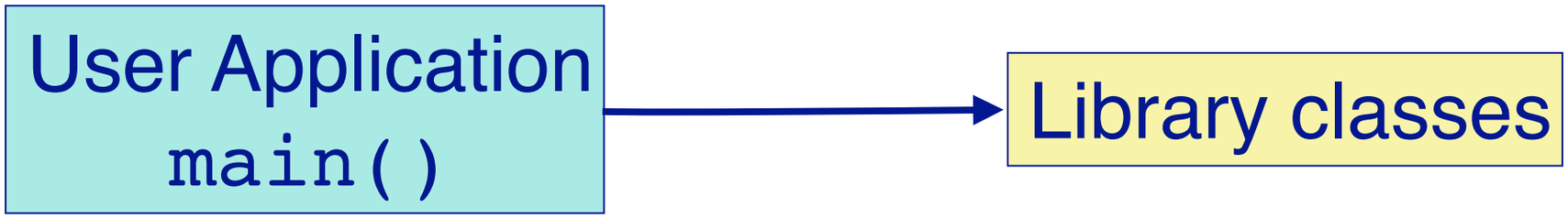
> *"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."*
>
> *Martin Fowler*

# JUnit - A Testing Framework

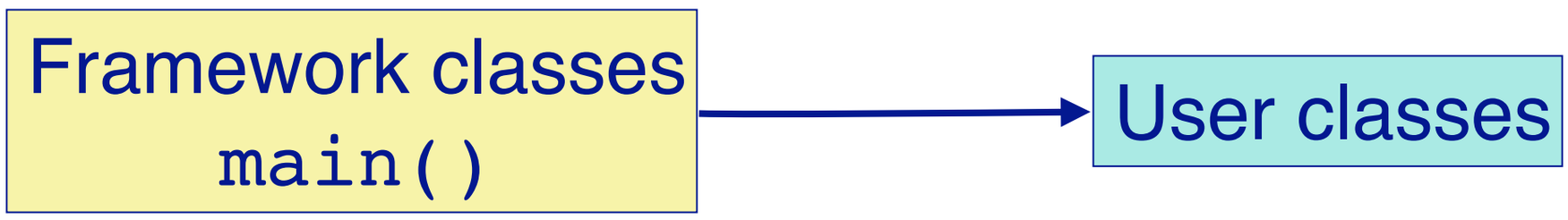
- JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks written by Kent Beck and Erich Gamma
- For documentation of how to use JUnit http://junit.sourceforge.net/doc/cookbook/cookbook.htm

# Frameworks vs. Libraries

In traditional application architectures, user code makes use of library functionality in the form of procedures or classes:




A framework reverses the usual relationship between generic and application code. Frameworks provide both generic functionality and application architecture:




*Essentially, a framework says:*
*"Don't call me — I'll call you."*

# JUnit 3.8

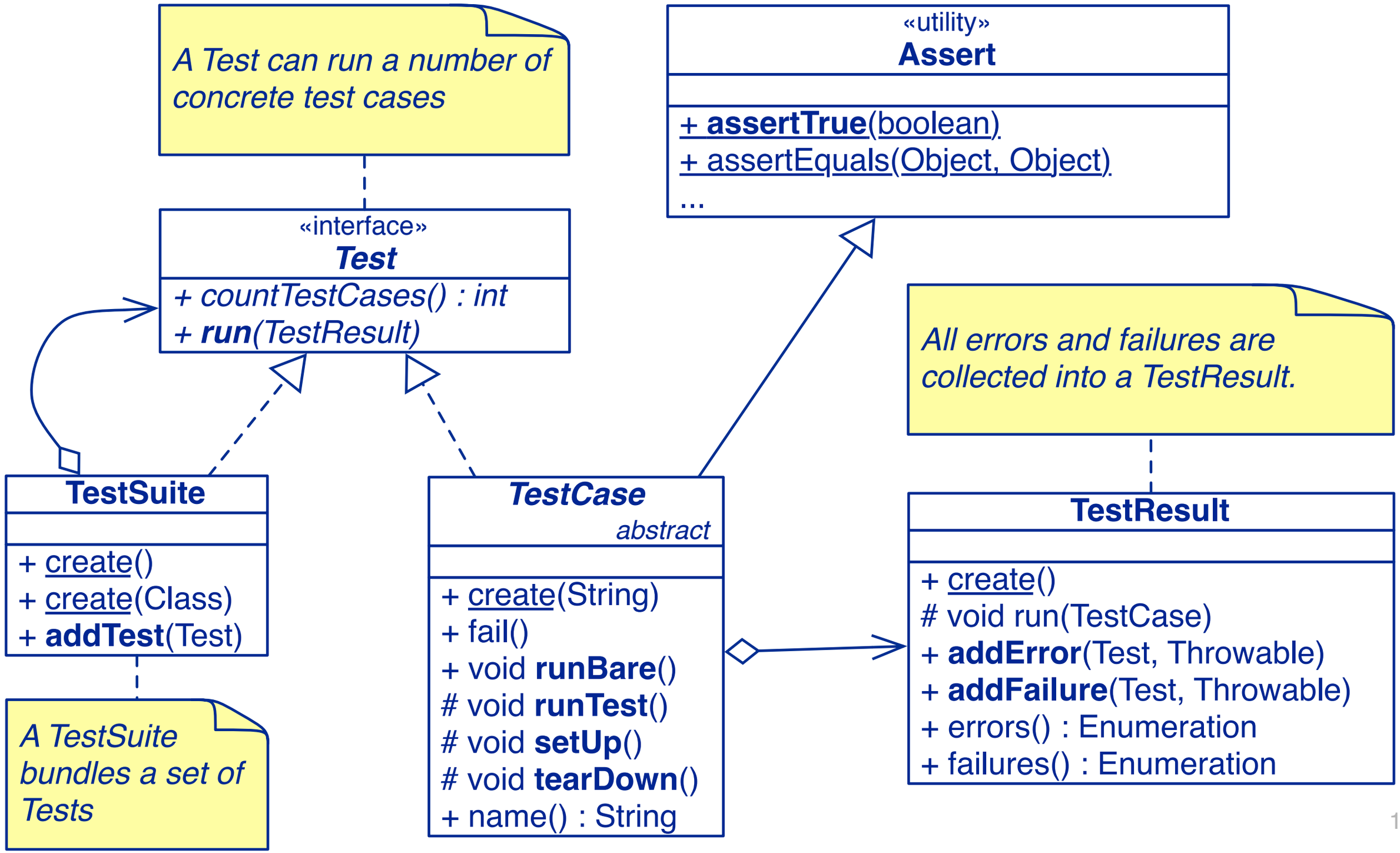
JUnit is a simple “testing framework” that provides:

- classes for writing *Test Cases and Test Suites*
- methods for *setting up and cleaning up test data* (“fixtures”)
- methods for *making assertions*
- textual and graphical tools for *running tests*

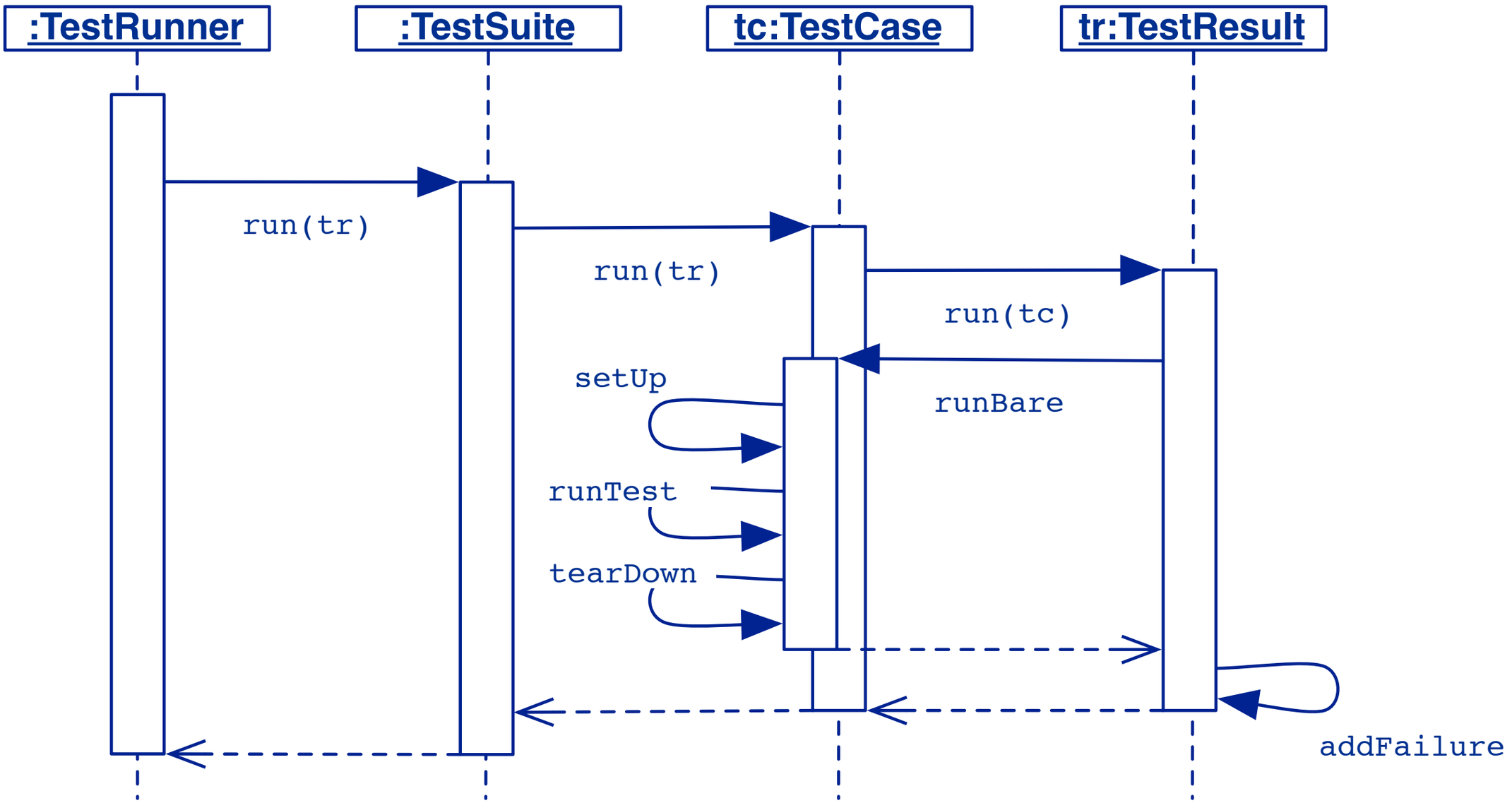
JUnit distinguishes between *failures and errors:*

- A failure is *a failed assertion*, i.e., an anticipated problem that you test.
- An error is *a condition you didn’t check for*, i.e., a runtime error.

# The JUnit 3.x Framework



# A Testing Scenario




*The framework calls the test methods that you define for your test cases.*

# JUnit 3.x Example Code

```
import junit.framework.*;
public class MoneyTest extends TestCase {
    private Money f12CHF;          // fixtures
    private Money f14CHF;

    protected void setUp() {       // create the test data
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
    }
    void testAdd() {               // create the test data
        Money expected = new Money(26, “CHF”);
        assertEquals(“amount not equal”,
                     expected,f12CHF.add(f14CHF);
    }
    ...
}
```

# Annotations in J2SE 5

- J2SE 5 introduces the Metadata feature (data about data)
- Annotations allow you to add decorations to your code (remember javadoc tags: *@author* )
- Annotations are used for code documentation, compiler processing (@Deprecated ), code generation, runtime processing

http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html

# JUnit 4.x

JUnit is a simple “testing framework” that provides:

- Annotations for marking methods as *tests*
- Annotations for marking methods that *setting up and cleaning up test data* (“fixtures”)
- methods for *making assertions*
- textual and graphical tools for *running tests*

# JUnit 4.x Example Code

```
import junit.framework.*;
import org.junit.*;
import static org.junit.Assert.*;
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;

    @Before public void setUp() {          // create the test data
       f12CHF = new Money(12, "CHF");      // - the fixture
       f14CHF = new Money(14, "CHF");
    }
    @Test public void testadd() {          // create the test data
       Money expected = new Money(26, “CHF”);
       assertEquals(“amount not equal”,
                    expected,f12CHF.add(f14CHF));
    }
    ...
}
```

# Testing Style

*“The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”*

- write unit tests that *thoroughly test a single class*
- write tests *as you develop* (even before you implement)
- write tests for *every new piece of functionality*

*“Developers should spend 25-50% of their time developing tests.”*

# Roadmap

- Junit — a testing framework
  - —Testing practices
  - —Frameworks vs. Libraries
  - —Junit 3.x vs. Junit 4.x (annotations)
- **Money and MoneyBag — a testing case study**
- Double Dispatch — how to add different types of objects
- JExample

# Representing multiple currencies

***The problem ...***

> *"The program we write will solve the problem of representing arithmetic with multiple currencies. Arithmetic between single currencies is trivial, you can just add the two amounts. ... Things get more interesting once multiple currencies are involved."*

# MoneyTest

We start by defining a *TestCase* that exercises the interface we would like our Money class to support:

```
import org.junit.*;
import static org.junit.Assert.*;
public class MoneyTest {
   private Money f12CHF;
   private Money f14CHF;

   @Before public void setUp() { // create the test data
      f12CHF = new Money(12, "CHF");
      f14CHF = new Money(14, "CHF");
   }
   ...
}
```

# Some basic tests

We define methods to test what we expect to be true ...

```
@Test public void testEquals() {
    assertNotNull(f12CHF);
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assertFalse(f12CHF.equals(f14CHF));
}

@Test public void testSimpleAdd() {
    Money expected = new Money(26, "CHF");
    Money result = f12CHF.add(f14CHF);
    assertEquals(expected, result);
}
```

***NB:*** `assertTrue`, etc. are static imported methods of the `Assert` class of the JUnit 4.x Framework and raise an `AssertionError` if they fail.

Junit 3.x raises a JUnit `AssertionFailedError` (!)

# Money

We now implement a Money class that fills our first few requirements:

```
public class Money {
   ...
   public Money add(Money m) {
      return new Money(...);
   }
   ...
}
```

| **Money** |
| --- |
| - fAmount : int<br>- fCurrency : String |
| + create(int, String)<br>+ amount() : int<br>+ currency() : String<br>+ add(Money) : Money<br>+ equals( Object) : boolean<br>+ toString() : String |

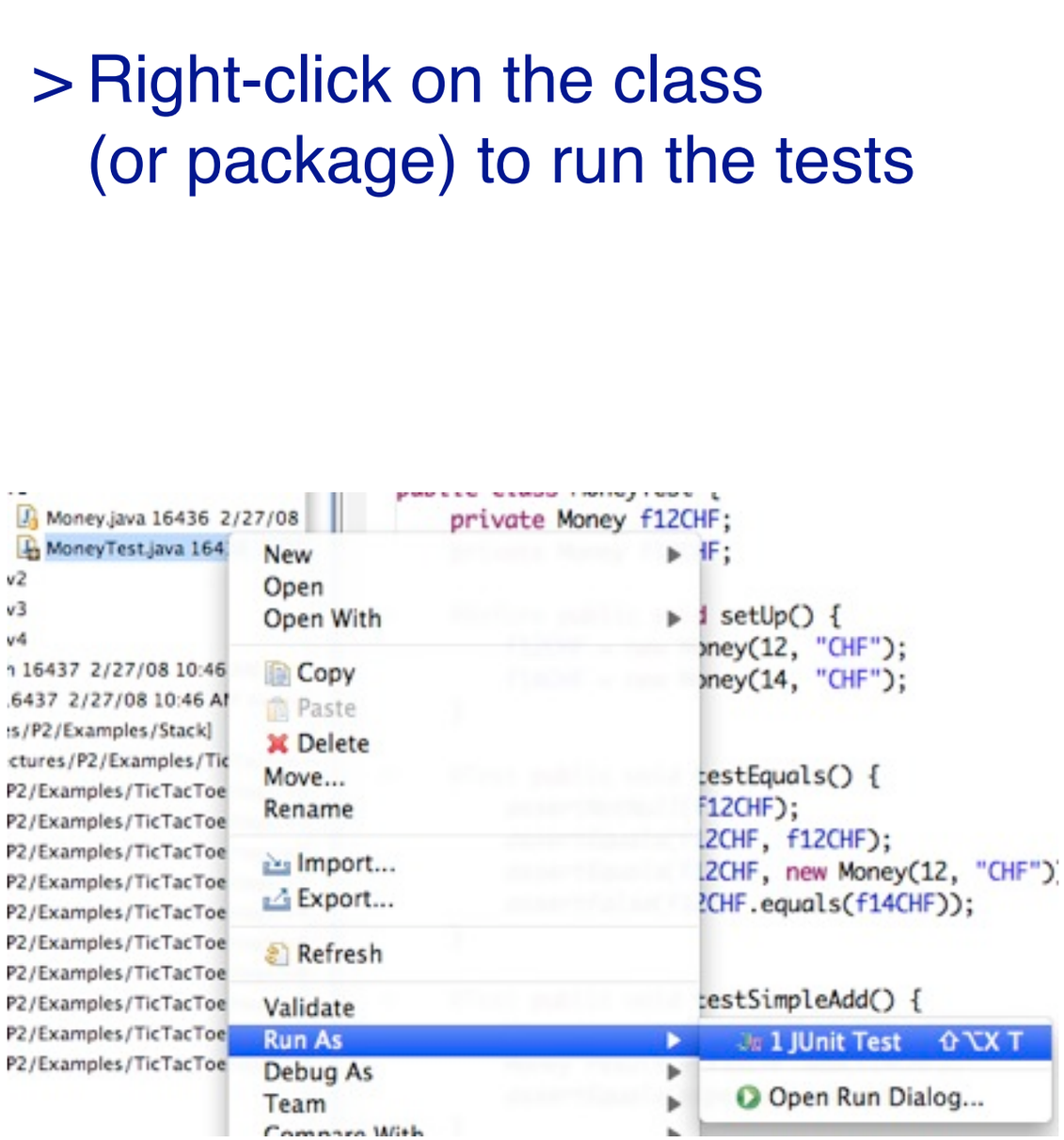
Note how the test case drives the design!

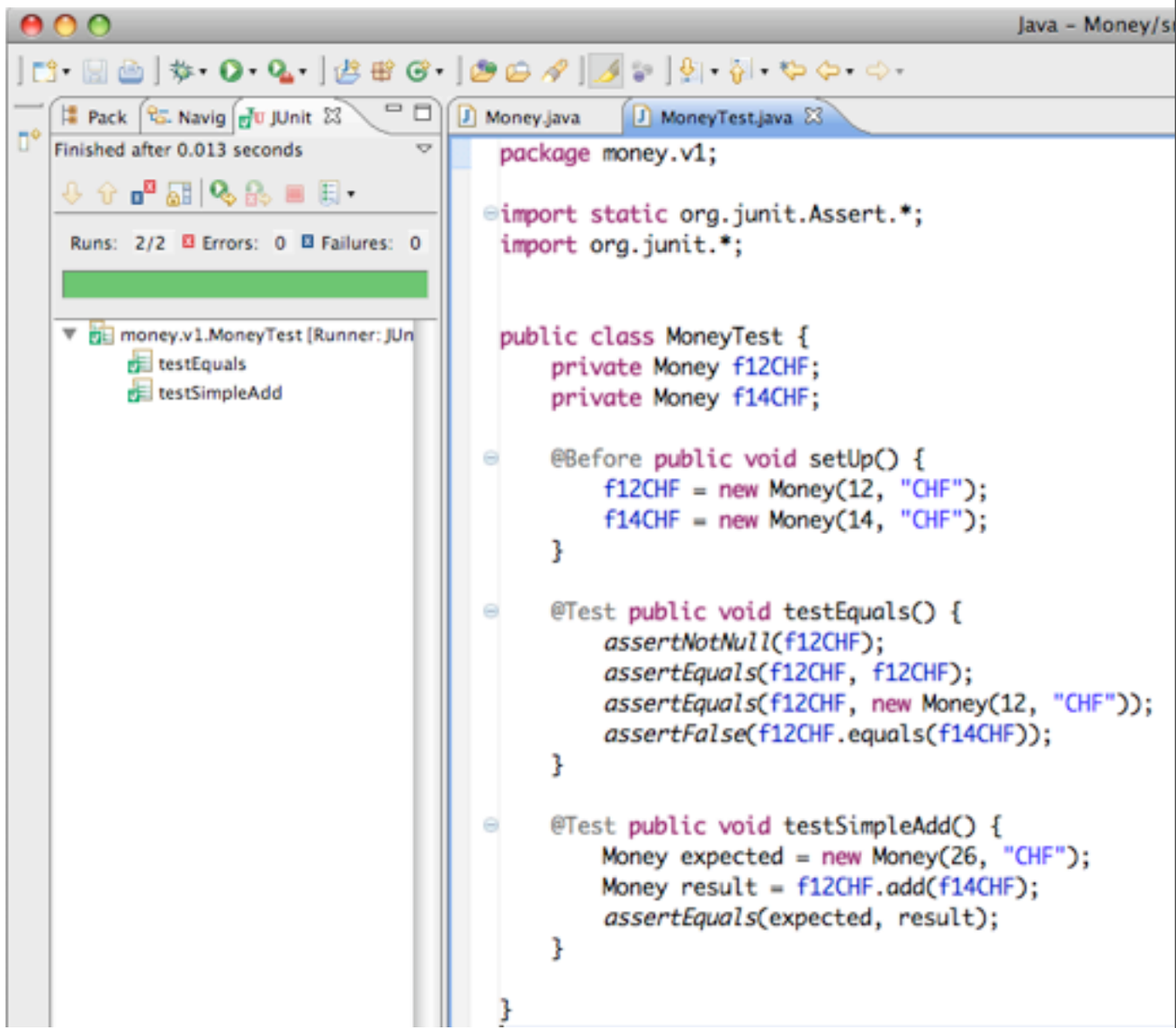
*NB: The first version does not consider how to add different currencies!*

*What should the class invariant be?*

# Running tests from eclipse

> Right-click on the class
(or package) to run the tests



# Testing MoneyBags (I)

To handle multiple currencies, we introduce a MoneyBag class that can hold several instances of Money:

```
import static org.junit.Assert.*;
public class MoneyTest {
    ...
    @Before public void setUp() {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
        f7USD = new Money( 7, "USD");
        f21USD = new Money(21, "USD");
        fMB1 = new MoneyBag(f12CHF, f7USD);
        fMB2 = new MoneyBag(f14CHF, f21USD);
    }
```

# Testing MoneyBags (II)

... and define some new (obvious) tests ...

```
@Test public void testBagEquals() {
    assertNotNull(fMB1);
    assertEquals(fMB1, fMB1);
    assertFalse(fMB1.equals(f12CHF));
    assertFalse(f12CHF.equals(fMB1));
    assertFalse(fMB1.equals(fMB2));
}
```

# MoneyBags

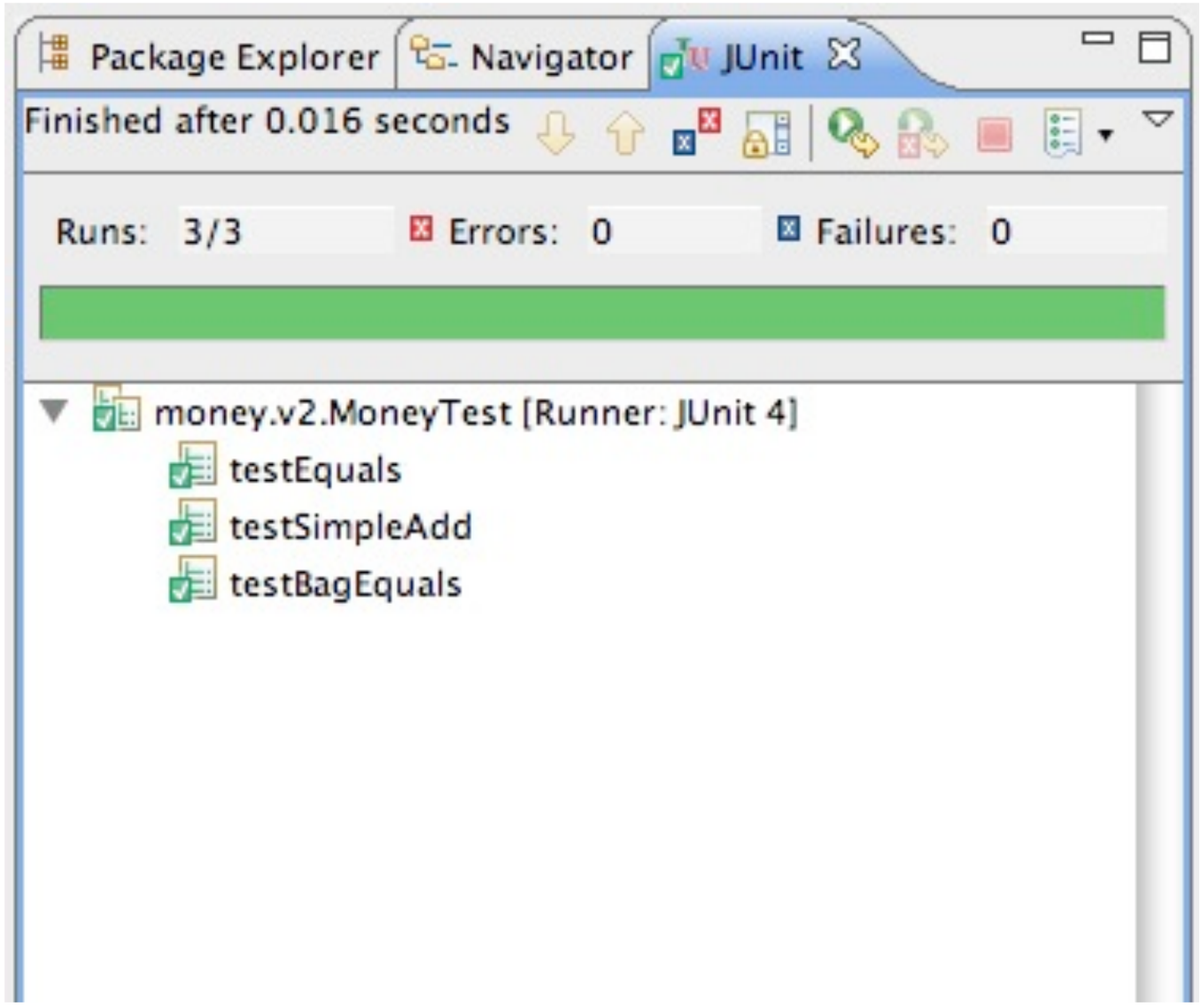
We can use a HashTable to keep track of multiple Monies:

```
class MoneyBag {
   private Hashtable monies = new Hashtable(5);
   MoneyBag(Money m1, Money m2) { ... }
   MoneyBag(Money bag[]) {
      for (Money money : bag) {
         appendMoney(money);
      }
   }
   private void appendMoney(Money aMoney) {
      Money m = (Money) monies.get(aMoney.currency());
      if (m != null) { m = m.add(aMoney); }
      else { m = aMoney; }
      monies.put(aMoney.currency(), m);
   }
}
```

| MoneyBag |
| --- |
| - fMonies : HashTable |
| + create(Money, Money)<br>+ create(Money [ ])<br>- appendMoney(Money)<br>+ toString() : String |

# Testing MoneyBags (III)

and we run the tests.



# Roadmap

- Junit — a testing framework
  - —Testing practices
  - —Frameworks vs. Libraries
  - —Junit 3.x vs. Junit 4.x (annotations)
- Money and MoneyBag — a testing case study
- **Double Dispatch — how to add different types of objects**
- JExample

# Adding MoneyBags

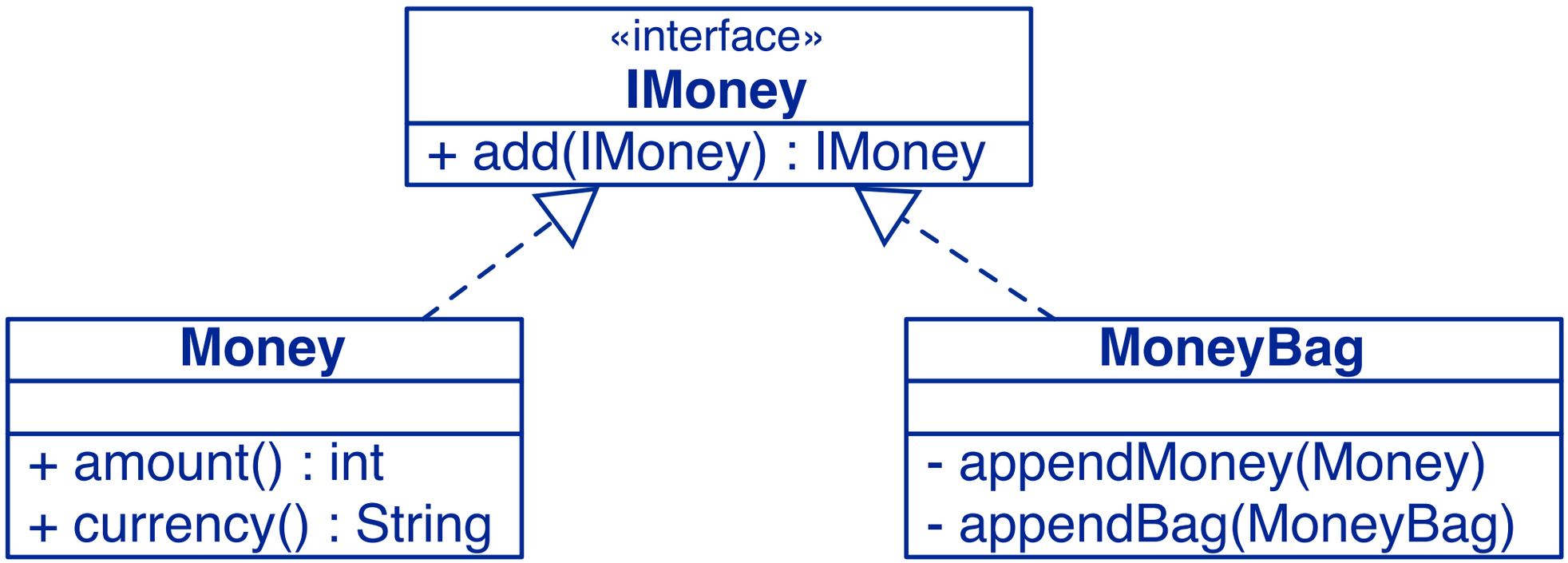
We would like to freely add together arbitrary Monies and MoneyBags, and be sure that *equals behave as equals*:

```
@Test public void mixedSimpleAdd() {
    // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
    Money bag[] = { f12CHF, f7USD };
    MoneyBag expected = new MoneyBag(bag);
    assertEquals(expected, f12CHF.add(f7USD));
}
```

*That implies that Money and MoneyBag should implement a common interface ...*

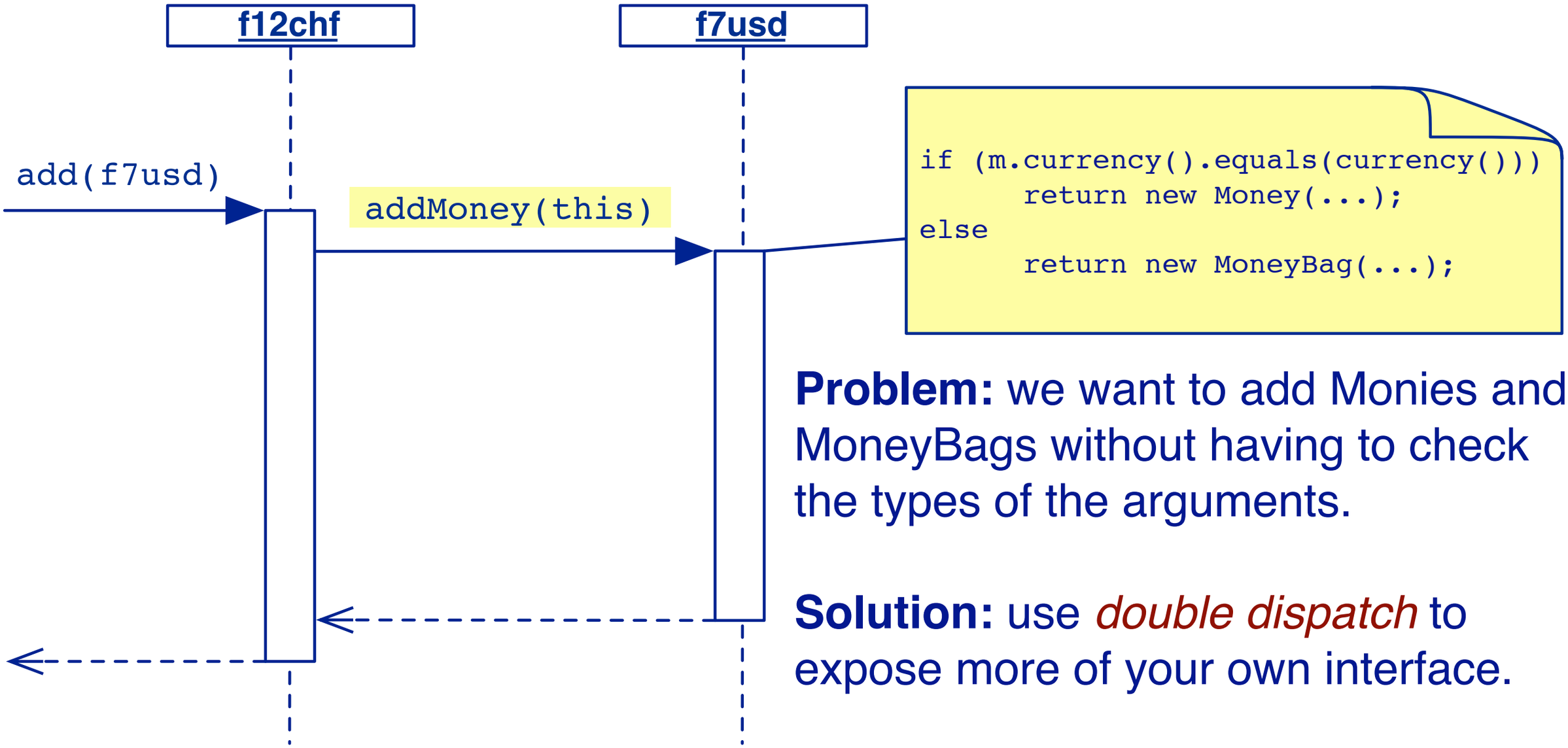
# The IMoney interface (I)

Monies know how to be added to other Monies




*Do we need anything else in the IMoney interface?*

# Double Dispatch (I)




**Problem:** we want to add Monies and MoneyBags without having to check the types of the arguments.

**Solution:** use *double dispatch* to expose more of your own interface.

# Double Dispatch (II)

*How do we implement add() without breaking encapsulation?*

```
class Money implements IMoney { ...
    public IMoney add(IMoney m) {
        return m.addMoney(this);          // add me as a Money
    } ...
}
class MoneyBag implements IMoney { ...
    public IMoney add(IMoney m) {
        return m.addMoneyBag(this);       // add as a MoneyBag
    } ...
}
```

*"The idea behind double dispatch is to use an additional call to discover the kind of argument we are dealing with..."*

# Double Dispatch (III)

The rest is then straightforward ...

```
class Money implements IMoney { ...
    public IMoney addMoney(Money m) {
        if (m.currency().equals(currency())) {
            return new Money(amount()+m.amount(),currency());
        }
        return new MoneyBag(this, m);
    }
    public IMoney addMoneyBag(MoneyBag s) {
        return s.addMoney(this);
    } ...
```

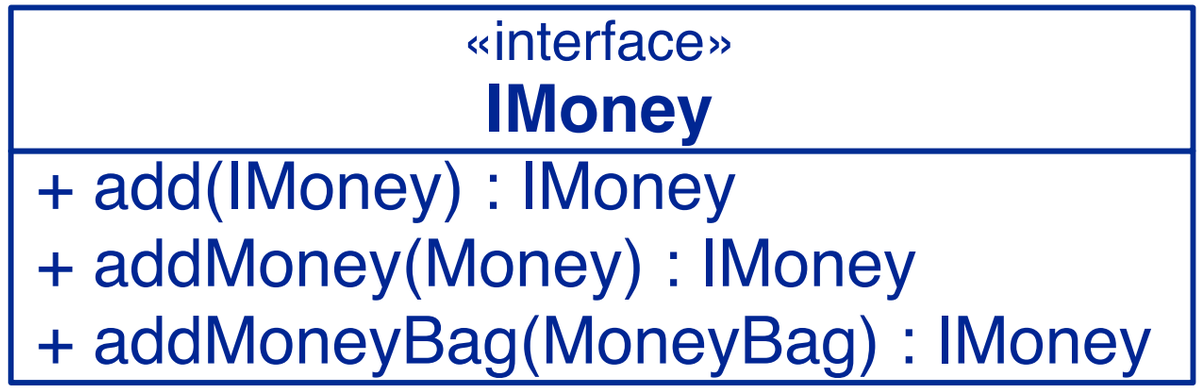
and MoneyBag takes care of the rest.

# Double Dispatch (IV)

- ***Pros:***
  - —No violation of encapsulation (no downcasting)
  - —Smaller methods; easier to debug
  - —Easy to add a new type
- ***Cons:***
  - —No centralized control
  - —May lead to an explosion of helper methods

# The IMoney interface (II)

So, the common interface has to be:

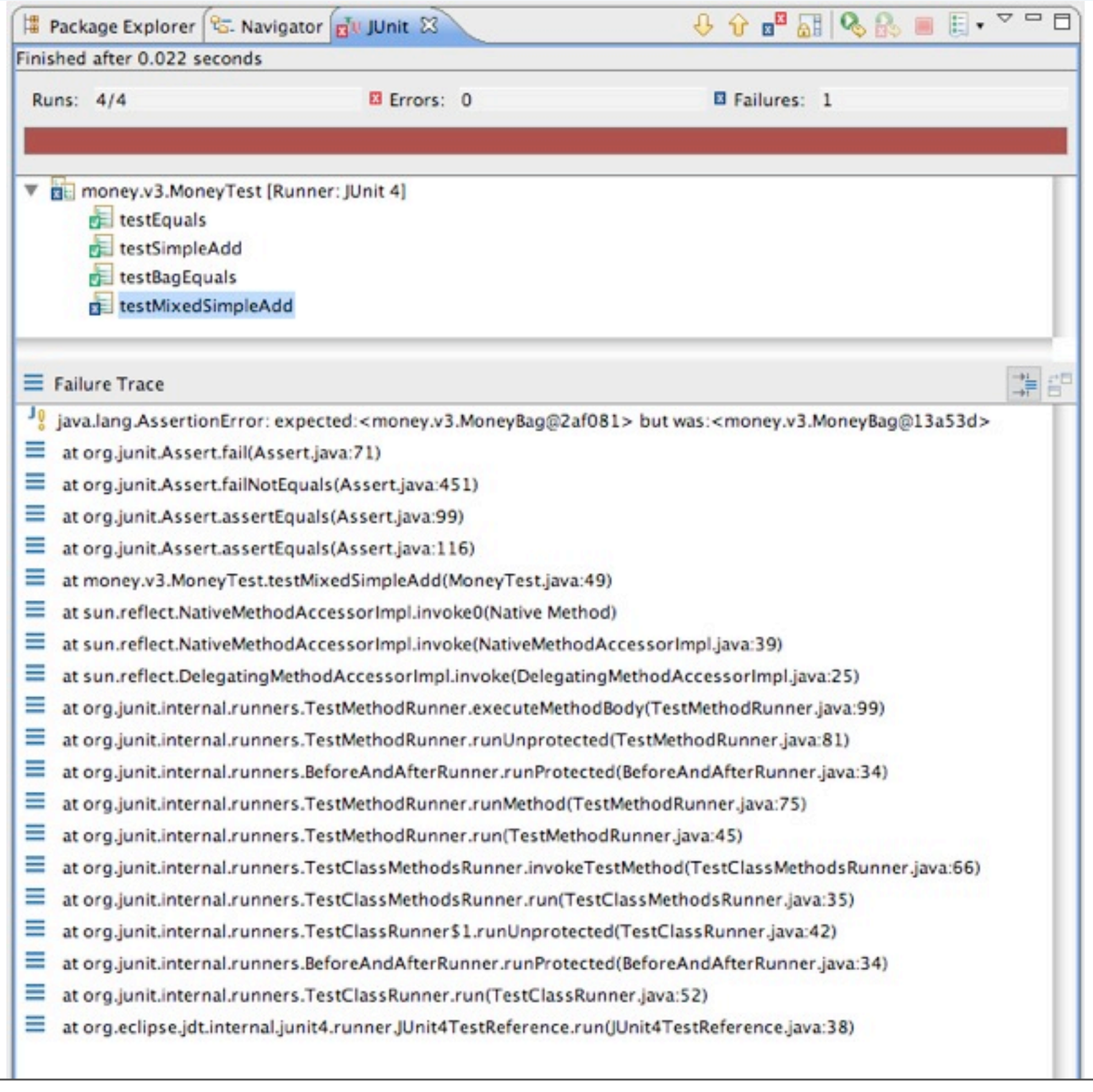



```
public interface IMoney {
    public IMoney add(IMoney aMoney);
    IMoney addMoney(Money aMoney);
    IMoney addMoneyBag(MoneyBag aMoneyBag);
}
```

*NB: addMoney() and addMoneyBag() are only needed within the Money package.*

# A Failed test

This time we are not so lucky ...

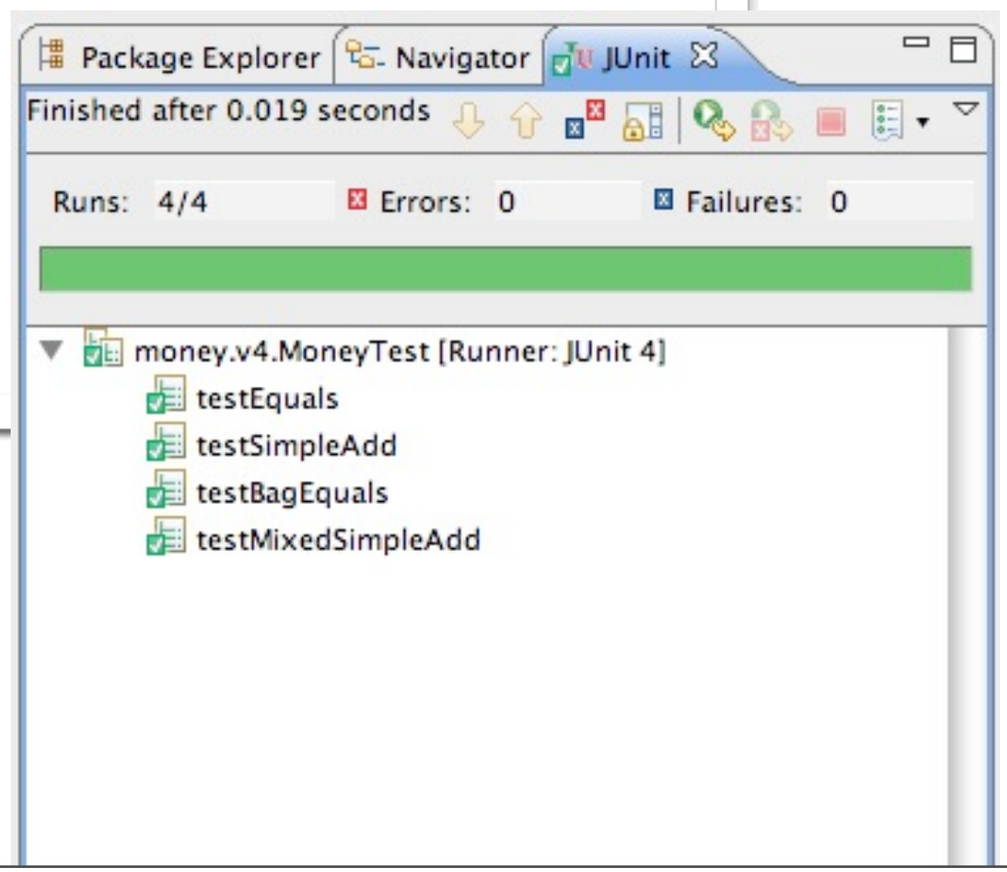


# The fix ...

It seems we forgot to implement MoneyBag.equals()!

*We fix it:*

```
class MoneyBag implements IMoney { ...
   public boolean equals(Object anObject) {
      if (anObject instanceof MoneyBag) {
         ...
      } else {
         return false;
      }
   }
```




... test it, and continue developing.

# Roadmap

- Junit — a testing framework
  - —Testing practices
  - —Frameworks vs. Libraries
  - —Junit 3.x vs. Junit 4.x (annotations)
- Money and MoneyBag — a testing case study
- Double Dispatch — how to add different types of objects
- **JExample**

# JExample

- JExample introduces *producer-consumer relationships* between tests
  - —Tests may *depend on* other tests that *produce examples* for them

http://scg.unibe.ch/Research/JExample/

# Stack example — imports

```
import java.util.Stack;
import java.util.EmptyStackException;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;

import ch.unibe.jexample.JExample;
import ch.unibe.jexample.Given;

@RunWith(JExample.class)
public class StackTest {
        …
}
```

# Stack example — dependencies

```
public class StackTest {

   @Test
   public Stack<String> empty() {
      Stack<String> stack = new Stack<String>();
      assertTrue(stack.empty());
      return stack;
   }

   @Test(expected=EmptyStackException.class)
   @Given("#empty")
   public void emptyPopFails(Stack<String> stack) {
      stack.pop();
   }
...
```

*Tests may return example objects*

*Consumer tests declare dependencies and arguments*

# Stack example — chained dependencies

```
public class StackTest {
...
   @Test
   @Given("#empty")
   public Stack<String> pushOnEmpty(Stack<String> stack) {
      stack.push("foo");
      assertFalse(stack.empty());
      assertTrue(stack.size() == 1);
      return stack;
   }
   @Test
   @Given("#pushOnEmpty")
   public Stack<String> pushPop(Stack<String> stack) {
      stack.pop();
      assertTrue(stack.empty());
      return stack;
   }
...
```

*Dependencies may be chained*

# Stack example — multiple dependencies

```
public class StackTest {
...
   @Test
   @Given("#pushPop; #empty")
   public void equality(Stack<String> used,
                        Stack<String> fresh) {
      assertEquals(used, fresh);
   }

}
```

*A test may depend on multiple tests*

# *What you should know!*

- *How does a framework differ from a library?*
- *What is a unit test?*
- *What is an annotation?*
- *How does JUnit 3.x differ from JUnit 4.x*
- *What is a test "fixture"?*
- *What should you test in a TestCase?*
- *How can testing drive design?*
- *What is "double dispatch"? What does the name mean?*

## *Can you answer these questions?*

- *How does implementing toString() help in debugging?*
- *How does the MoneyTest suite know which test methods to run?*
- *How does the TestRunner invoke the right suite() method?*
- *Why doesn't the Java compiler complain that MoneyBag.equals() is used without being declared?*