A Testing Framework (Part 2/2)

Alexandre Bergel
http://bergel.eu
20-09-2021

Last time…

We implemented two classes: Money, MoneyBag, and MoneyTest

We had a falling test, which is

```
@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));
}
```

Outline for today

1. Double dispatch - how to add different types of objects

2. Exercise: Catchipun

Outline for today

1. Double dispatch - how to add different types of objects
2. Exercise: Catchipun

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 ...

Adding MoneyBags

```
f12CHF.add(f7USD) “=> return a money bag”

new MoneyBag().add(f12CHF) “=> return a money bag”

f12CHF.add(f12CHF) “=> return a money”

f12CHF.add(new MoneyBag()) “=> return a money bag”

…
```

A possible solution

```
public class Money {
  public Object add(Object m) {
    if (m instanceof Money) { ... }
    if (m instanceof MoneyBag) { ... }
    // error here?
  }
}
```

```
public class MoneyBag {
  public Object add(Object m) {
    if (m instanceof Money) { ... }
    if (m instanceof MoneyBag) { ... }
    // error here?
  }
}
```

A possible solution

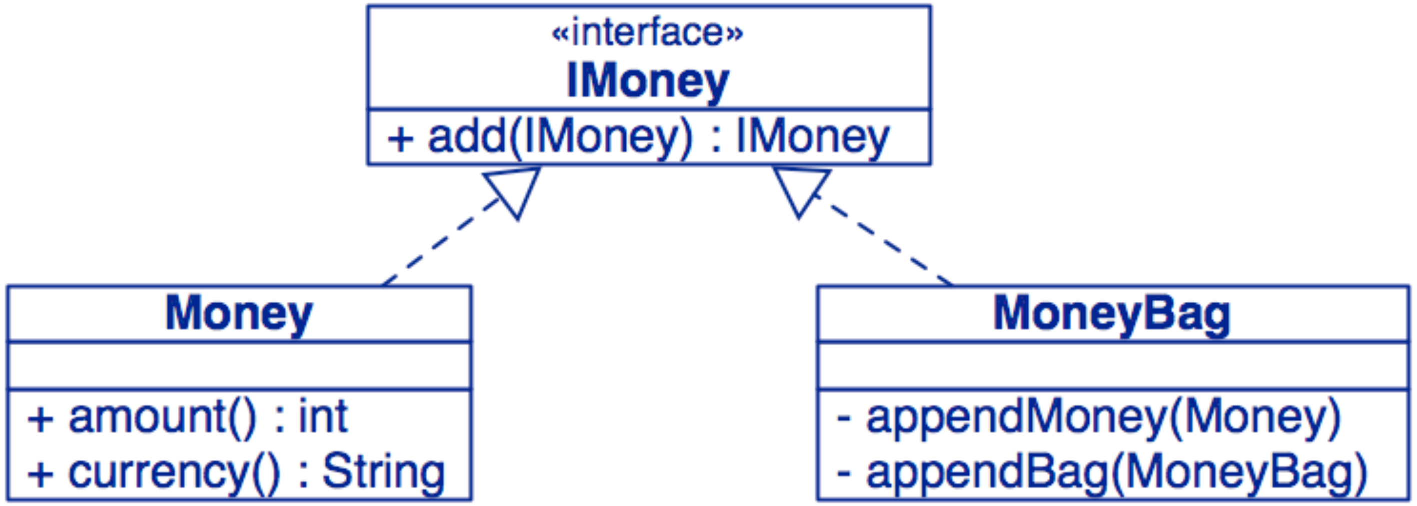
```
public class Money {
  public Object add(Object m) {
    if (m instanceof Money) { ... }
    if (m instanceof MoneyBag) { ... }
    // error here?
  }
}
```

no no, we do not want that!

```
 ublic class MoneyBag {
    lic Object add(Object m) {
      (m instanceof Money) { ... }
      (m instanceof MoneyBag) { ... }
     / error here?
  }
}
```

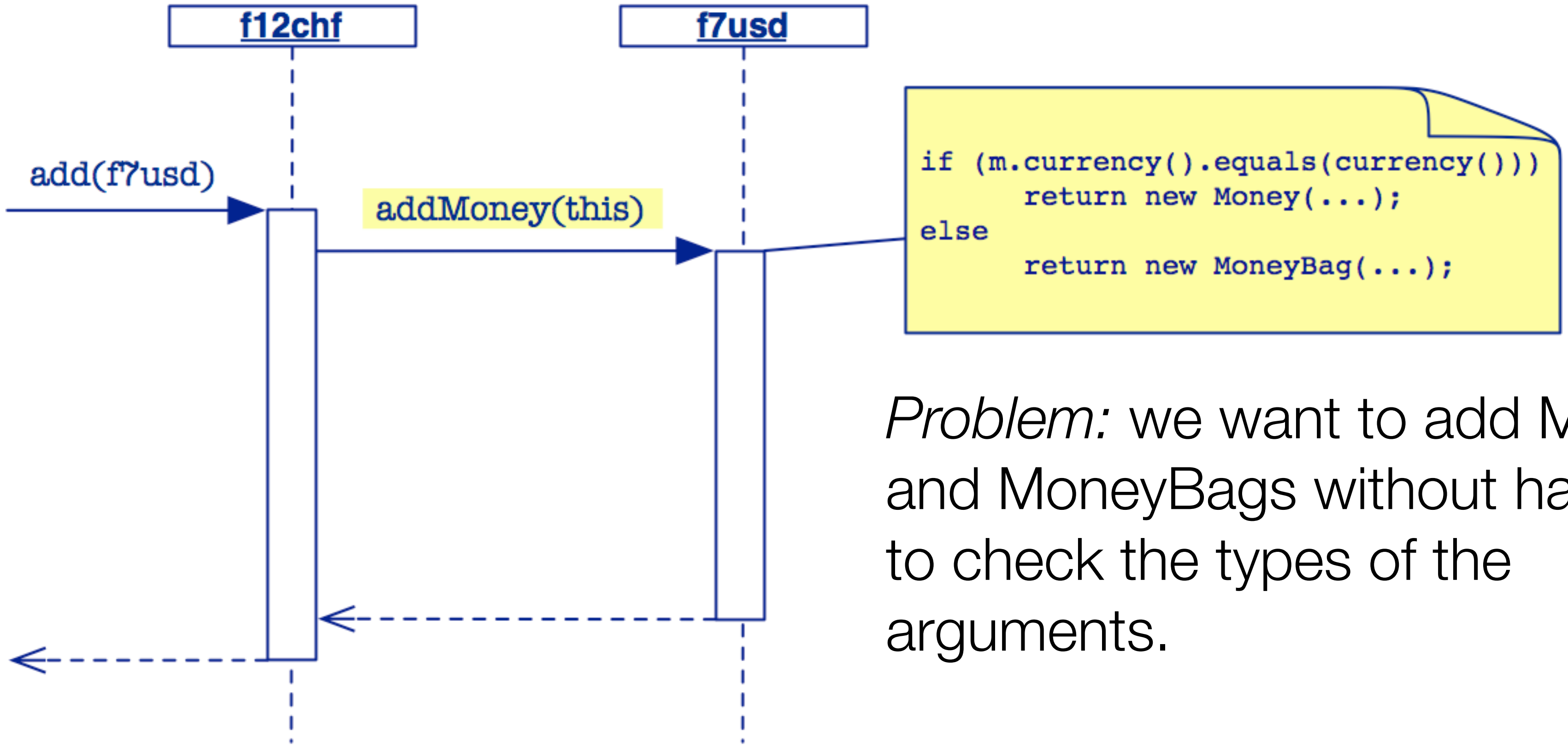
The IMoney interface (I)

Monies know how to be added to other Monies



[NOTE: The diagram is incomplete, we will complete it later on]

Double Dispatch (I)



```
if (m.currency().equals(currency()))
      return new Money(...);
else
      return new MoneyBag(...);
```

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());
      } else {
         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

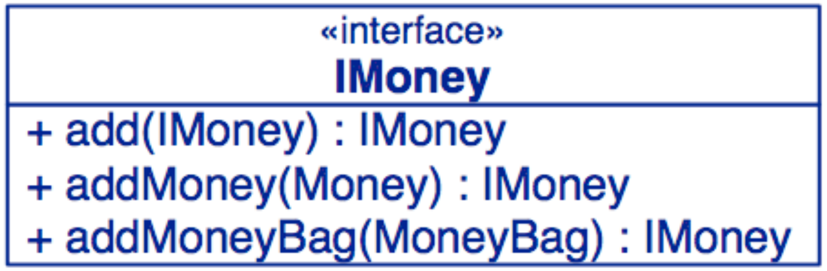
- No violation of encapsulation (no downcasting)
- Smaller methods; easier to debug
- Easy to add a new type

Cons

- No centralized control
- May lead to an explosion of helper methods

The IMoney interface (II)

So, the common interface has to be:

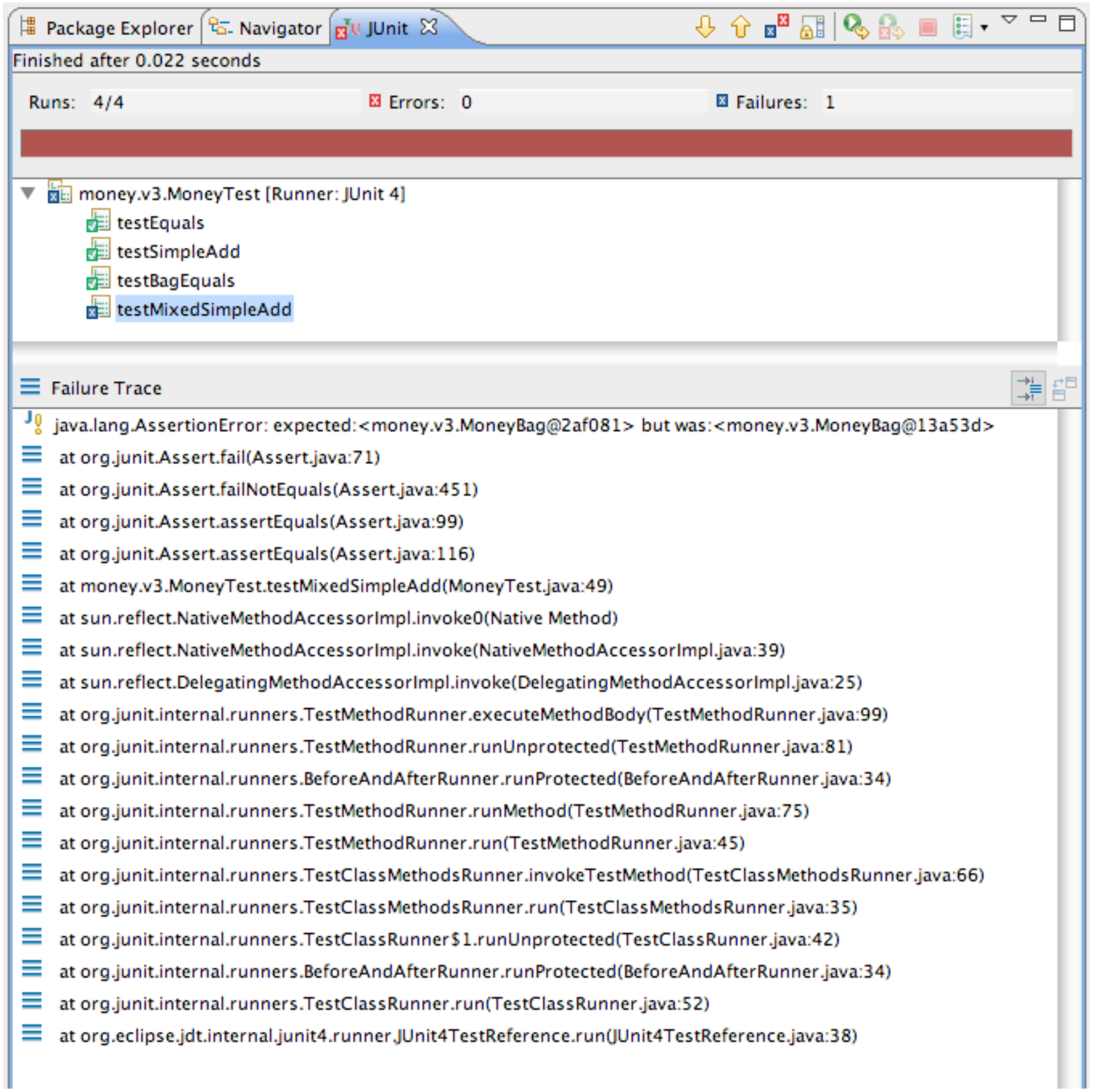


```
public interface IMoney {
   public IMoney add(IMoney aMoney);
   IMoney addMoney(Money aMoney);
   IMoney addMoneyBag(MoneyBag aMoneyBag);
}
```

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

A Failed test

This time we are not so lucky ...



Finished after 0.022 seconds

Runs: 4/4   Errors: 0   Failures: 1

money.v3.MoneyTest [Runner: JUnit 4]
- testEquals
- testSimpleAdd
- testBagEquals
- testMixedSimpleAdd

Failure Trace

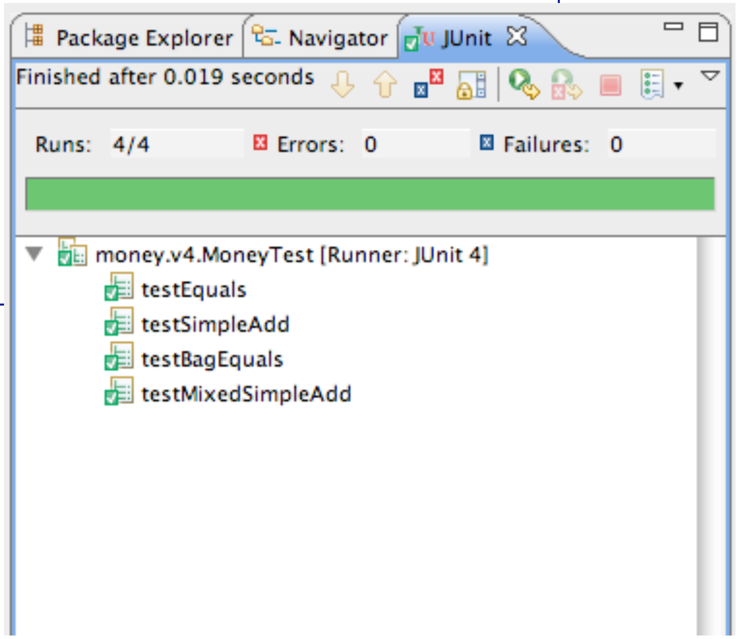
```
java.lang.AssertionError: expected:<money.v3.MoneyBag@2af081> but was:<money.v3.MoneyBag@13a53d>
at org.junit.Assert.fail(Assert.java:71)
at org.junit.Assert.failNotEquals(Assert.java:451)
at org.junit.Assert.assertEquals(Assert.java:99)
at org.junit.Assert.assertEquals(Assert.java:116)
at money.v3.MoneyTest.testMixedSimpleAdd(MoneyTest.java:49)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at org.junit.internal.runners.TestMethodRunner.executeMethodBody(TestMethodRunner.java:99)
at org.junit.internal.runners.TestMethodRunner.runUnprotected(TestMethodRunner.java:81)
at org.junit.internal.runners.BeforeAndAfterRunner.runProtected(BeforeAndAfterRunner.java:34)
at org.junit.internal.runners.TestMethodRunner.runMethod(TestMethodRunner.java:75)
at org.junit.internal.runners.TestMethodRunner.run(TestMethodRunner.java:45)
at org.junit.internal.runners.TestClassMethodsRunner.invokeTestMethod(TestClassMethodsRunner.java:66)
at org.junit.internal.runners.TestClassMethodsRunner.run(TestClassMethodsRunner.java:35)
at org.junit.internal.runners.TestClassRunner$1.runUnprotected(TestClassRunner.java:42)
at org.junit.internal.runners.BeforeAndAfterRunner.runProtected(BeforeAndAfterRunner.java:34)
at org.junit.internal.runners.TestClassRunner.run(TestClassRunner.java:52)
at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:38)
```

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;
      }
   }
```



Finished after 0.019 seconds

Runs: 4/4   Errors: 0   Failures: 0

money.v4.MoneyTest [Runner: JUnit 4]
testEquals
testSimpleAdd
testBagEquals
testMixedSimpleAdd

... test it, and continue developing.

Outline for today

1. Double dispatch - how to add different types of objects
2. Exercise: Catchipun

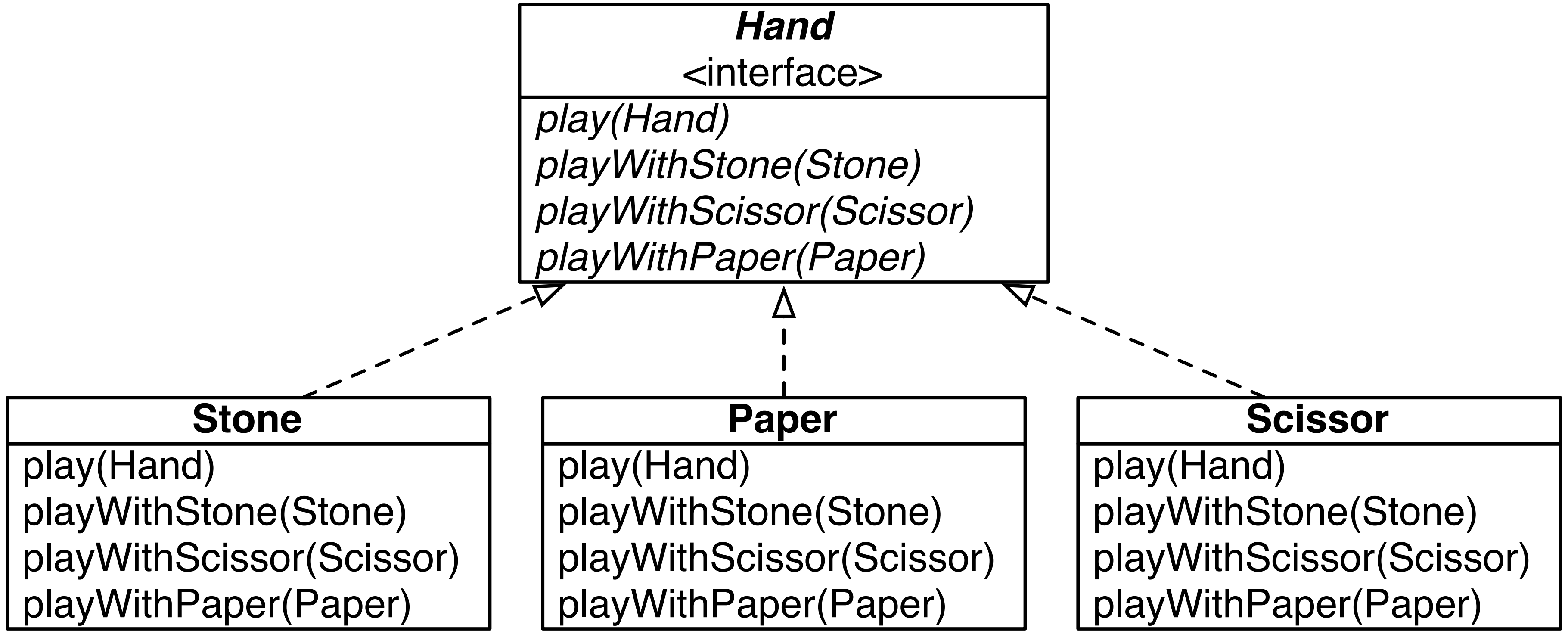
CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Cachipun

Though it looks simple, designing this small game is a fantastic example of the double dispatch design pattern

This pattern is particularly important since it is the base of many other design patterns

Design



Hand
<interface>
play(Hand)
playWithStone(Stone)
playWithScissor(Scissor)
playWithPaper(Paper)
Stone
play(Hand)
playWithStone(Stone)
playWithScissor(Scissor)
playWithPaper(Paper)
Paper
play(Hand)
playWithStone(Stone)
playWithScissor(Scissor)
playWithPaper(Paper)
Scissor
play(Hand)
playWithStone(Stone)
playWithScissor(Scissor)
playWithPaper(Paper)

```
interface Hand {
   // 1 win, 0 draw, -1 loose
   int play (Hand v);
   int playWithStone (Stone stone);
   int playWithPaper (Paper paper);
   int playWithScissor (Scissor scissor);
}
```

```
class Stone implements Hand {
  public int play(Hand v) {
    return v.playWithStone (this);
  }
  public int playWithStone (Stone v) {
    return 0;
  }
  public int playWithScissor (Scissor v) {
    return -1;
  }
  public int playWithPaper (Paper v) {
    return 1;
  }
}
```

```java
class Paper implements Hand {
  public int play(Hand v) {
    return v.playWithPaper (this);
  }
  public int playWithStone (Stone v) {
    return -1;
  }
  public int playWithScissor (Scissor v) {
    return 1;
  }
  public int playWithPaper (Paper v) {
    return 0;
  }
}
```

```
class Scissor implements Hand {
  public int play(Hand v) {
    return v.playWithScissor (this);
  }
  public int playWithStone (Stone v) {
    return 1;
  }
  public int playWithScissor (Scissor v) {
    return 0;
  }
  public int playWithPaper (Paper v) {
    return -1;
  }
}
```

Benefit of using double dispatch

Methods are shorts

Methods do not contains “`if`” and “`instanceof`”

This means that code is *easier to test*, thanks to double dispatch

Ideally, `instanceof` has to be used only in the `equals` method

The cost of adding a new type (e.g., spoke or 🤘) is very low

What you should know

How does the double dispatch pattern work?

When should one apply this pattern?

What are the benefits when using it?

Can you answer these questions?

Can you give an example where the double dispatch is successfully employed?

Can the double dispatch be used to always get rid of the if statements?

License

dcc
CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE