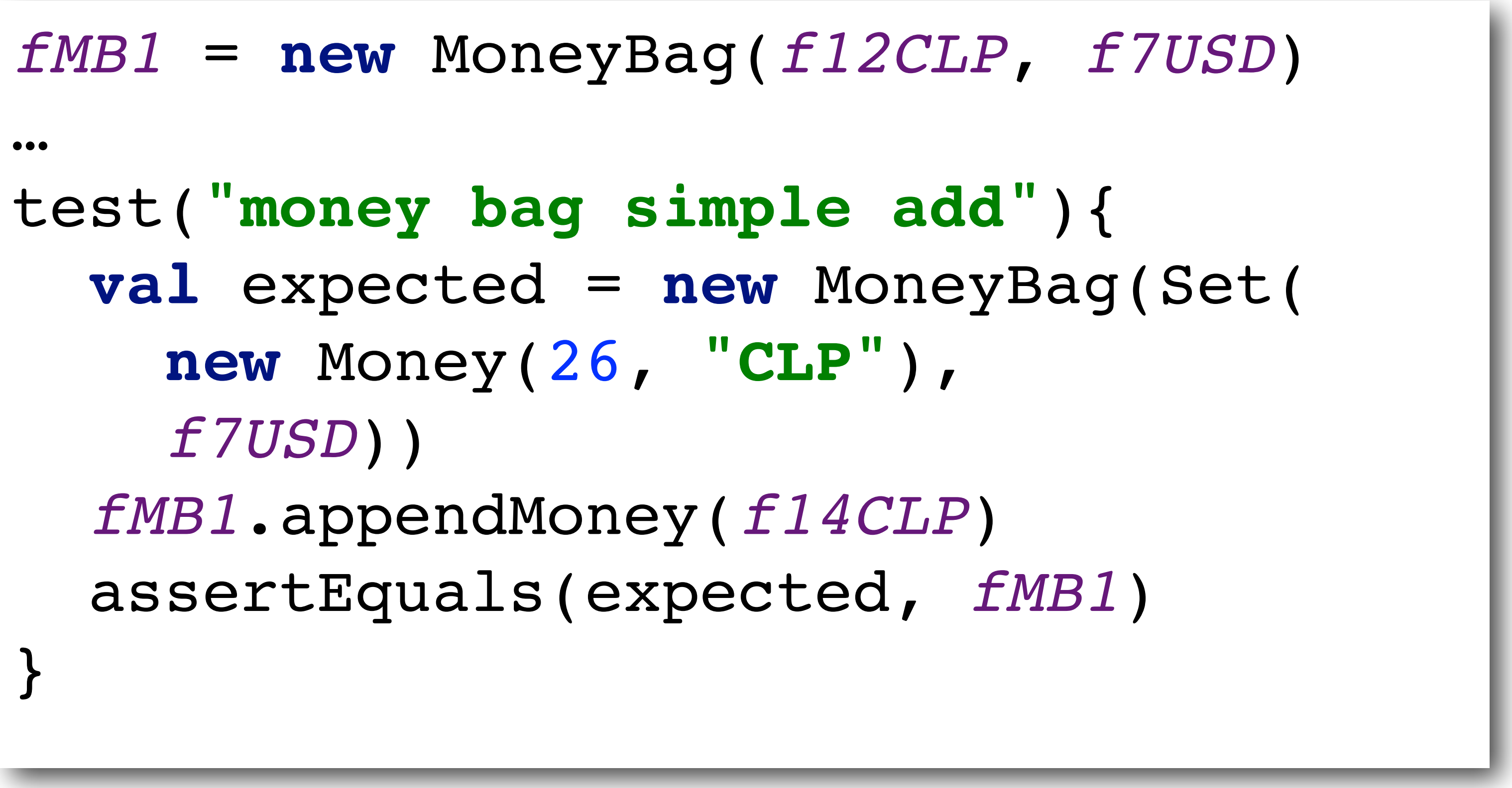
Double Dispatch (Testing II)

Nancy Hitschfeld
Matías Toro

Last time…

We implemented two classes: Money, MoneyBag, and MoneyTest

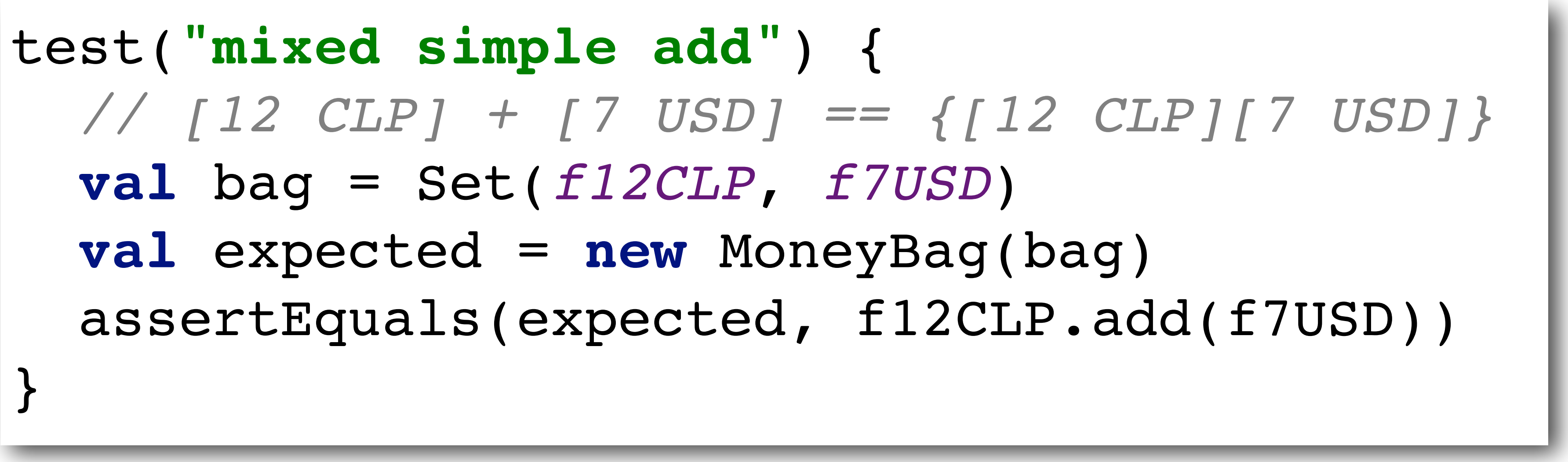


```
fMB1 = new MoneyBag(f12CLP, f7USD)
…
test("money bag simple add"){
  val expected = new MoneyBag(Set(
    new Money(26, "CLP"),
    f7USD))
  fMB1.appendMoney(f14CLP)
  assertEquals(expected, fMB1)
}
```

Last time...

We implemented two classes: Money, MoneyBag, and MoneyTest

We had a falling test, which is



```
test("mixed simple add") {
  // [12 CLP] + [7 USD] == {[12 CLP][7 USD]}
  val bag = Set(f12CLP, f7USD)
  val expected = new MoneyBag(bag)
  assertEquals(expected, f12CLP.add(f7USD))
}
```

Outline for today

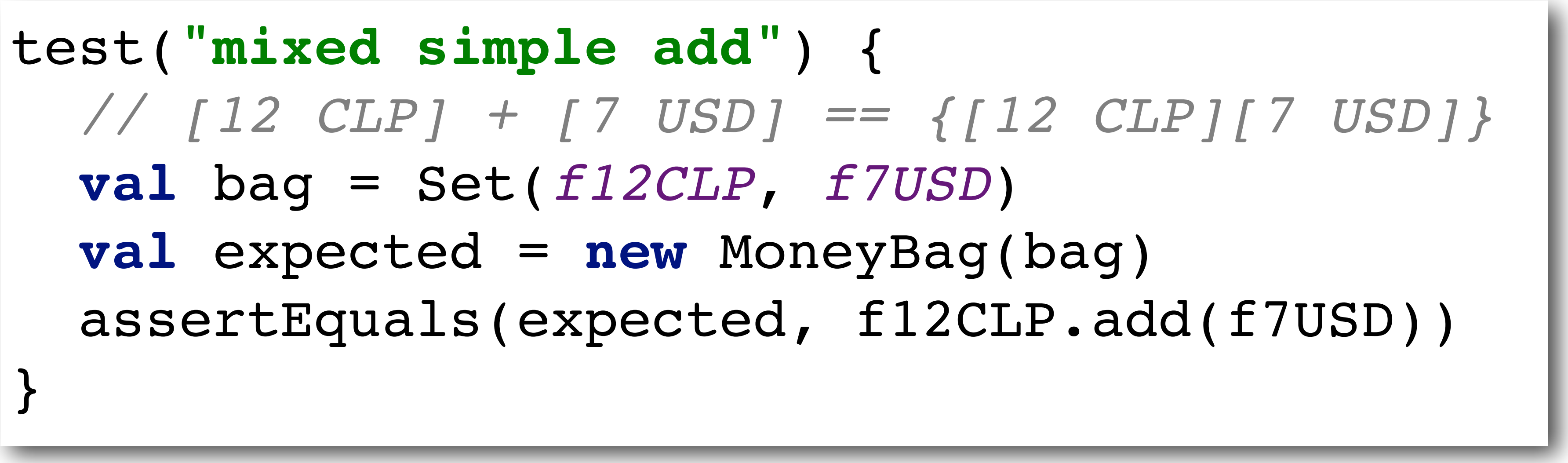
1. Double dispatch - how to add different types of objects
2. Exercise: Cachipun

Outline for today

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

Adding MoneyBags

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



```
test("mixed simple add") {
  // [12 CLP] + [7 USD] == {[12 CLP][7 USD]}
  val bag = Set(f12CLP, f7USD)
  val expected = new MoneyBag(bag)
  assertEquals(expected, f12CLP.add(f7USD))
}
```

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

Adding MoneyBags

```
f12CLP.add(f7USD) “=> return a money bag”


new MoneyBag().add(f12CLP) “=> return a money bag”


f12CLP.add(f12CLP) “=> return a money”


f12CLP.add(new MoneyBag()) “=> return a money bag”

…
```

A possible solution

```
class Money {
  def add(m: Any): Any = {
    if (m.isInstanceOf[Money]) { ... }
    if (m.isInstanceOf[MoneyBag]) { ... }
    // error here?
  }
}
```

```
class MoneyBag {
  def add(m: Any): Any = {
    if (m.isInstanceOf[Money]) { ... }
    if (m.isInstanceOf[MoneyBag]) { ... }
    // error here?
  }
}
```

A possible solution

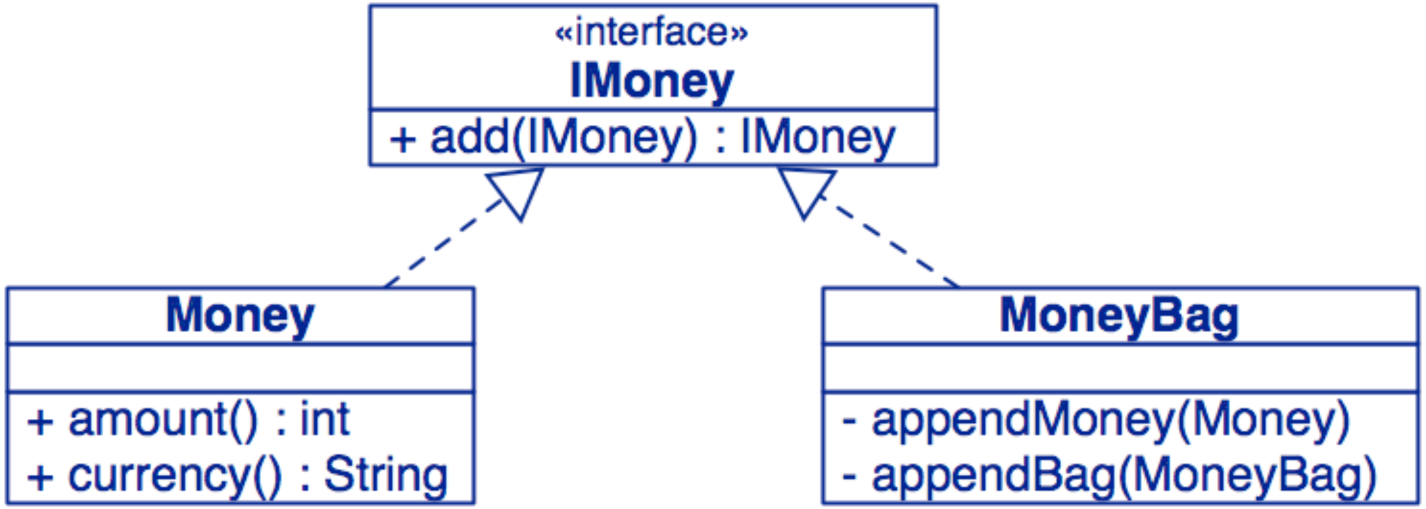
```
class Money {
  def add(m: Any): Any = {
    if(m.isInstanceOf[Money]) { ... }
    if (m.isInstanceOf[MoneyBag]) { ... }
    // error here?
  }
}
```

```
class MoneyBag {
  def a                     Any = {
                               ney]) { ... }
                               eyBag]) { ... }



}
```

no no, we do not want that!

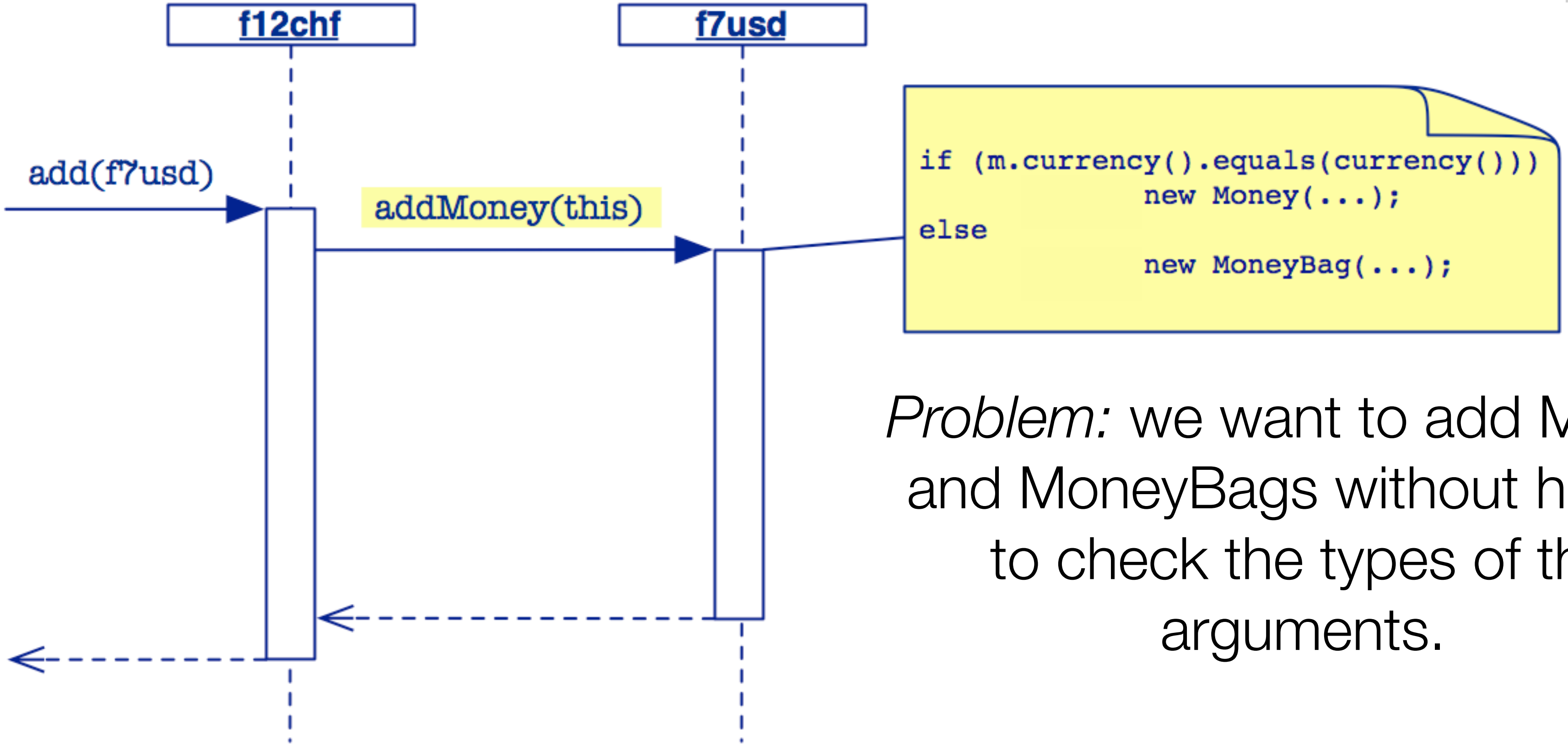
The IMoney interface (I)

Monies know how to be added to other Monies



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

Double Dispatch (I)



f12chf

f7usd

add(f7usd)

addMoney(this)

```
if (m.currency().equals(currency()))
                new Money(...);
else
                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 extends IMoney{
  override def add(m: IMoney): IMoney = {
    m.addMoney(this)
  } …
}
class MoneyBag extends IMoney{
  override def add(m: IMoney): IMoney = {
    m.addMoneyBag(this)
  } …
}
```

Add me as Money

Add me as 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(…) extends IMoney{
  def addMoney(m: Money): IMoney = {
    if(m.currency.equals(currency))
      new Money(value+m.value, currency)
    else
      new MoneyBag(this, m)
  }
  def addMoneyBag(mb: MoneyBag): IMoney = {
    mb.addMoney(this)
  } …
}
```

and MoneyBag takes care of the rest.

Double Dispatch (IV)

Pros

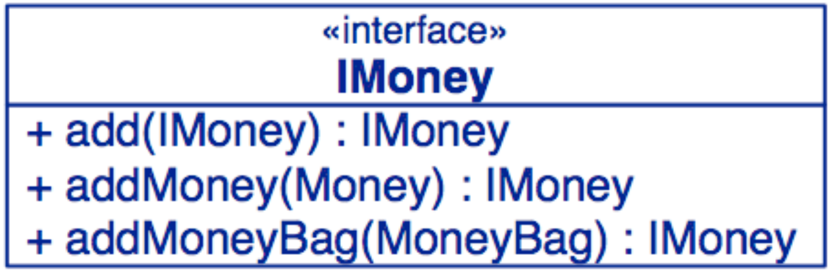
- No violation of encapsulation (no downcasting)
- Smaller methods; easier to debug
- Easy to add a new type

Cons

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

The IMoney interface (II)

So, the common interface has to be:



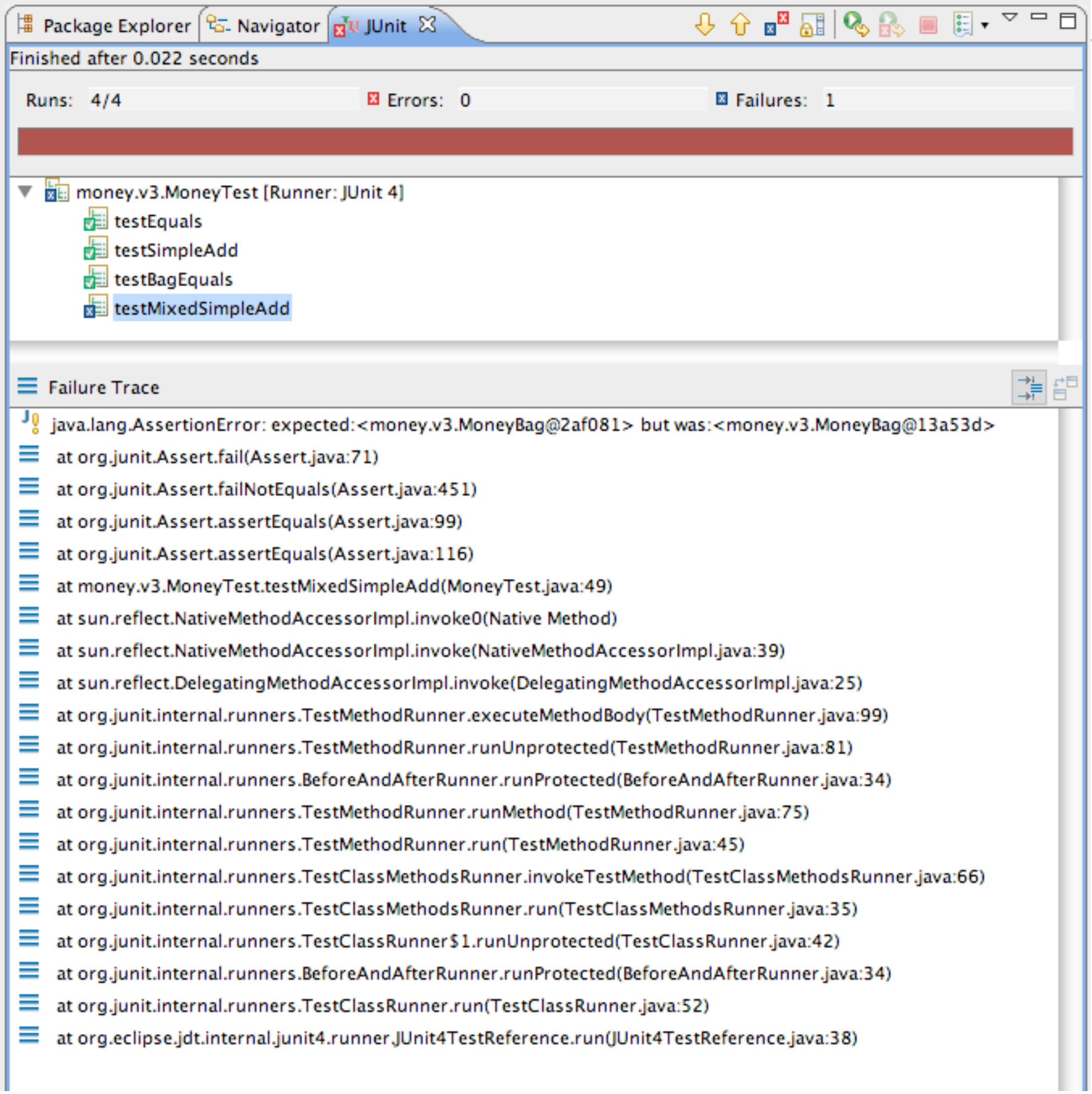
«interface»
IMoney
+ add(IMoney) : IMoney
+ addMoney(Money) : IMoney
+ addMoneyBag(MoneyBag) : IMoney

```
trait IMoney {
  def add(m: IMoney): IMoney
  def addMoney(m: Money): IMoney
  def addMoneyBag(mb: MoneyBag): IMoney
}
```

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

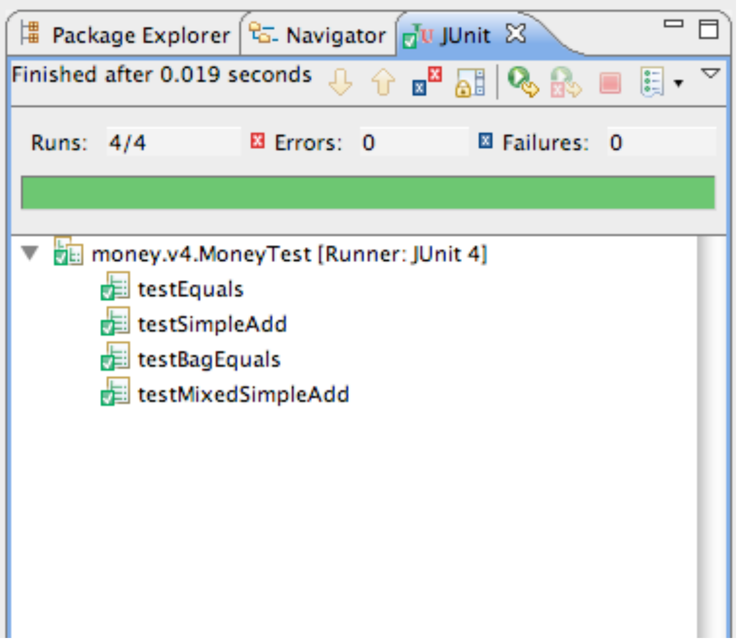
```
java.lang.AssertionError: expected:<money.v3.MoneyBag@2af081> but was:<money.v3.MoneyBag@13a53d>
 at org.junit.Assert.fail(Assert.java:71)
 at org.junit.Assert.failNotEquals(Assert.java:451)
 at org.junit.Assert.assertEquals(Assert.java:99)
 at org.junit.Assert.assertEquals(Assert.java:116)
 at money.v3.MoneyTest.testMixedSimpleAdd(MoneyTest.java:49)
 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 extends IMoney{
  …
  override def equals(other: Any): Boolean
    (other.getClass.getName == getClass.get
      monies.equals(other.asInstanceOf[Mone
  }
}
```



```
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

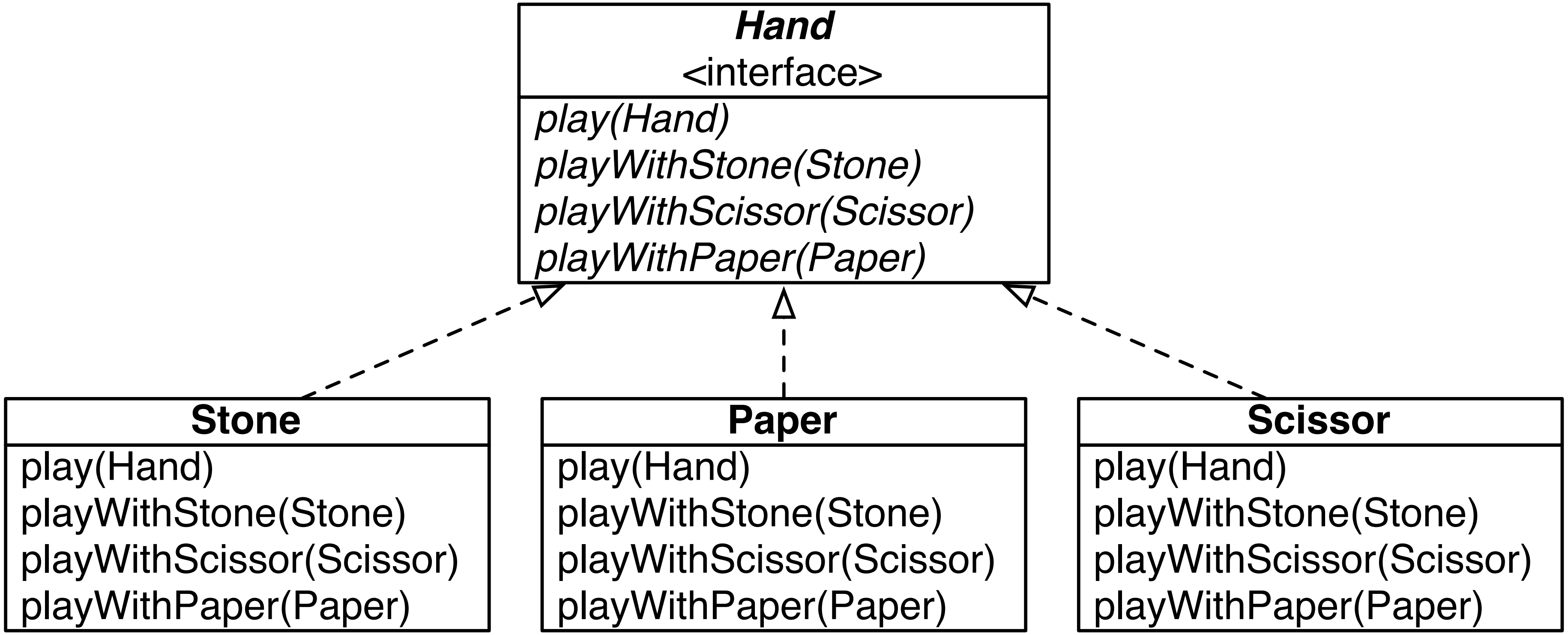
1. Double dispatch - how to add different types of objects
2. Exercise: Cachipun

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)

```scala
trait Hand {
  // 1 win, 0 draw, -1 loose
  def play(v: Hand): Int
  def playWithStone(stone: Stone): Int
  def playWithPaper(paper: Paper): Int
  def playWithScissor(scissor: Scissor): Int
}
```

```
class Stone extends Hand {
  def play(v: Hand): Int = v.playWithStone(this)

  def playWithStone(stone: Stone): Int = 0

  def playWithPaper(paper: Paper): Int = 1

  def playWithScissor(scissor: Scissor): Int = -1
}
```

```scala
class Paper extends Hand {
  def play(v: Hand): Int = v.playWithPaper(this)

  def playWithStone(stone: Stone): Int = -1

  def playWithPaper(paper: Paper): Int = 0

  def playWithScissor(scissor: Scissor): Int = 1
}
```

```
class Scissor extends Hand{
  def play(v: Hand): Int = v.playWithScissor(this)

  def playWithStone(stone: Stone): Int = 1

  def playWithPaper(paper: Paper): Int = -1

  def playWithScissor(scissor: Scissor): Int = 0
}
```

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

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)
You are free to:
- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms

Attribution: you must give appropriate credit

ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: https://creativecommons.org/licenses/by-sa/4.0/

(Diapositivas de Alex Bergel)

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE