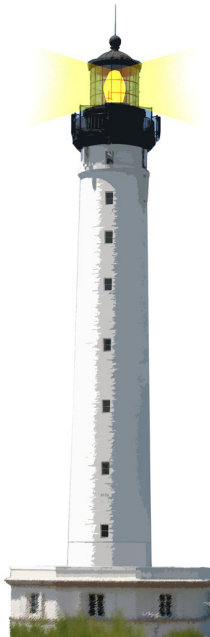
Object-Oriented
Design Lecture

Stone Paper Scissors - J version

Stéphane Ducasse

http://stephane.ducasse.free.fr
http://car.mines-douai.fr/luc



Pharo

http://www.pharo.org

Let us start with a test

```
@Test
  public void testStoneVsStone() {
    assertEquals(new Stone().play(new Stone()),"draw");
    }
```

Now the Stone class...

```
class Stone {
  public String play (Stone h){
    return ...
  }
}
```

A hint

No there is no need for conditionals.

A second hint

- Sending a message is making a choice
- When writing a method I know the class of the message receiver. Dull not quite!

So Stone...

```
class Stone {
  public String play (Stone h){
    return h.playAgainstStone(this);
  }
}
```

Now playAgainstStone ...

```
class Stone {
  public String play (Stone h){
    return h.playAgainstStone(this);
  }
  public String playAgainstStone(Stone s){
    return ...
  }
}
```

Stone...

```
class Stone {
  public String play (Stone h){
    return h.playAgainstStone(this);
  }
  public String playAgainstStone(Stone s){
    return "draw";
  }
}
```

Another test...

```
@Test
  public void testStoneVsPaper() {
    assertEquals(new Stone().play(new Paper()),"paper");
    }
```

Paper...

```
class Paper {
  public String playAgainstStone(Paper s){
    return "paper";
  }
}
```

Well this is Java, ...

- The previous code cannot compile because Paper and Stone are unrelated
- All the types should be known
- All the methods should be reachable statically

Two solutions:

- Common superclass
- Using interfaces

Defining and using an interface

```
interface IHand {
  String play (IHand h);
  String playAgainstStone (IHand s);
  String playAgainstScissors (IHand s);
  String playAgainstPaper (IHand p);
}
```

Stone...

```
class Stone {
  public String play (IHand h){
    return h.playAgainstStone(this);
  }
  public String playAgainstStone(IHand s){
    return "draw";
  }
}
```

Full Solution: Stone

```
class Stone implements IHand {
  public String play (IHand h){
    return h.playAgainstStone(this);
  }
  public String playAgainstStone(IHand s){ return "draw";}
  public String playAgainstScissors(IHand s){ return "stone";}
  public String playAgainstPaper(IHand s){ return "paper";}}
```

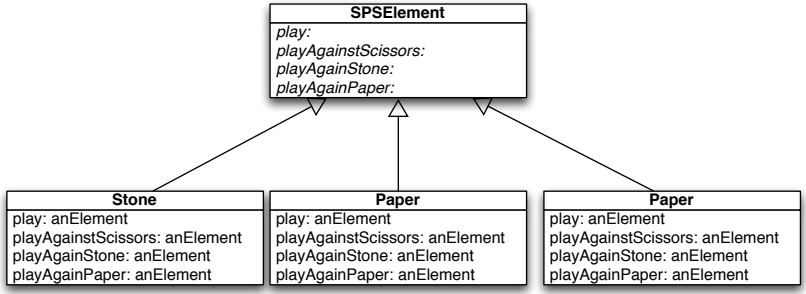
Full Solution: Scissors

```
class Scissors implements IHand {
   public String play (IHand h){
      return h.playAgainstScissors(this);
   }
   public String playAgainstStone(IHand s){ return "stone";}
   public String playAgainstScissors(IHand s){ return "draw";}
   public String playAgainstPaper(IHand s){ return "scissors";}
}
```

Full Solution: Paper

```
class Paper implements IHand {
   public String play (IHand h){
      return h.playAgainstPaper(this);}
   public String playAgainstStone(IHand s){ return "paper";}
   public String playAgainstScissors(IHand s){ return "scissors";}
   public String playAgainstPaper(IHand s){ return "draw";}
}
```

Another possible solution



Remark

In this example we do not need to pass the argument during the double dispatch. Usually double dispatch

- uses arguments
- is between more classes (document elements and operations)

Conclusion

- Powerful
- Modular
- Just sending an extra message to an argument and using late binding

A course by

Stéphane Ducasse
http://stephane.ducasse.free.fr

and

Luc Fabresse
http://car.mines-douai.fr/luc