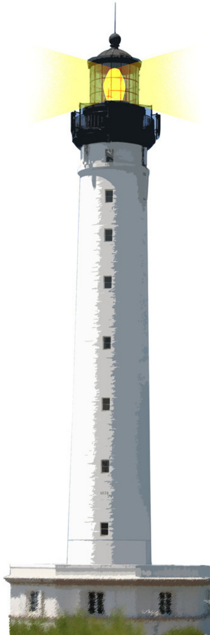
Advanced Object-Oriented Design Lecture

Stone Paper Scissors

and a cool sub



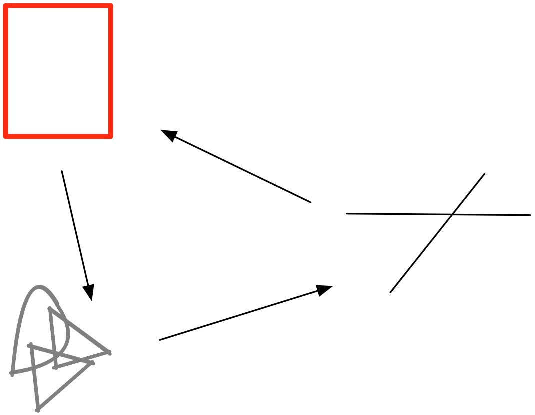
http://www.pharo.org

Goals

- In the quest of dispatch
- No conditionals!

```
(Stone new vs: Paper new)
>>> #paper
```

Goals



Stone Paper Scissors: one Test

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Stone new vs: Paper new) equals: #paper
```

The inverse too

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Stone new vs: Paper new) equals: #paper
```

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Paper new vs: Stone new) equals: #paper
```

More Tests

```
StonePaperScissorsTest >> testStoneAgainsStone
  self assert: (Stone new vs: Stone new) equals: #draw
```

```
StonePaperScissorsTest >> testStoneIsWinning
  self assert: (Stone new vs: Scissors new) equals: #stone
```

Let us start

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Stone new vs: Paper new) equals: #paper
```

```
Stone >> vs: anotherTool
  ^ ...
```

Hints

- The solution does not contain an explicit condition
- Remember sending a message is making a choice
- Sending a message is selecting the right method
  - when we send the message vs: the method of the receiver is executed
- What if we introduce another method?

Paper playAgainstStone:

```
Stone >> vs: anotherTool
  ^ anotherTool playAgainstStone: self
```

```
Paper >> playAgainstStone: aStone
  ^ ...
```

Paper playAgainstStone:

```
Stone >> vs: anotherTool
  ^ anotherTool playAgainstStone: self
```

```
Paper >> playAgainstStone: aStone
>> ^ #paper
```

Paper playAgainstStone:

Works for

```
Stone new vs: Paper new
>>> #paper
```

But not for

```
Stone new vs: Scissor new
>>> #stone
```

- How to fix this?
- Easy!

Other playAgainstStone:

```
Scissors >> playAgainstStone: aStone
  ^ #stone
```

```
Stone >> playAgainstStone: aStone
  ^ #draw
```

Stepping back

- We know that a method is executed on a class (here Stone)
- We SEND another message to the argument to select another method (here playAgainstStone:)
- We sent two messages to be able to select a method based on its receiver AND argument

Scissors now

```
Scissors >> vs: anotherTool
  ^ anotherTool playAgainstScissors: self
```

```
Scissors >> playAgainstScissors: aScissors
  ^ #draw
```

```
Paper >> playAgainstScissors: aScissors
  ^ #scissors
```

```
Stone >> playAgainstScissors: aScissors
  ^ #stone
```

Paper now

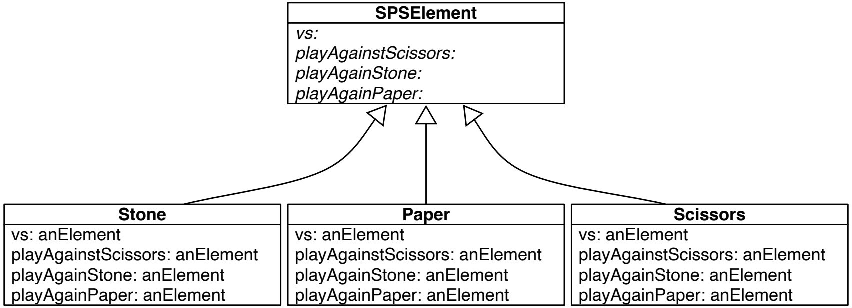
```
Paper >> vs: anotherTool
  ^ anotherTool playAgainstPaper: self
```

```
Scissors >> playAgainstPaper: aPaper
  ^ #scissors
```

```
Paper >> playAgainstPaper: aPaper
  ^ #draw
```

```
Stone >> playAgainstPaper: aPaper
  ^ #paper
```

Solution Overview


SPSElement
vs:
playAgainstScissors:
playAgainStone:
playAgainPaper:
Stone
vs: anElement
playAgainstScissors: anElement
playAgainStone: anElement
playAgainPaper: anElement
Paper
vs: anElement
playAgainstScissors: anElement
playAgainStone: anElement
playAgainPaper: anElement
Scissors
vs: anElement
playAgainstScissors: anElement
playAgainStone: anElement
playAgainPaper: anElement

```
Paper >> vs: anotherTool
        ^ anotherTool playAgainstPaper: self
```

Remark

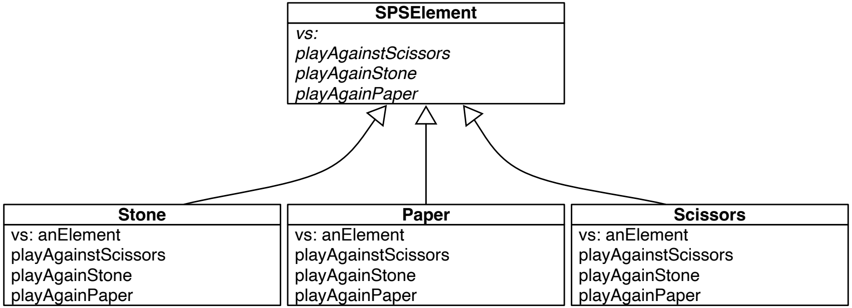
- In this toy example we do not need to pass the argument during the double dispatch
- But in general this is important as in Visitor

```
Scissors >> playAgainstPaper: aPaper
  ^ #scissors
```

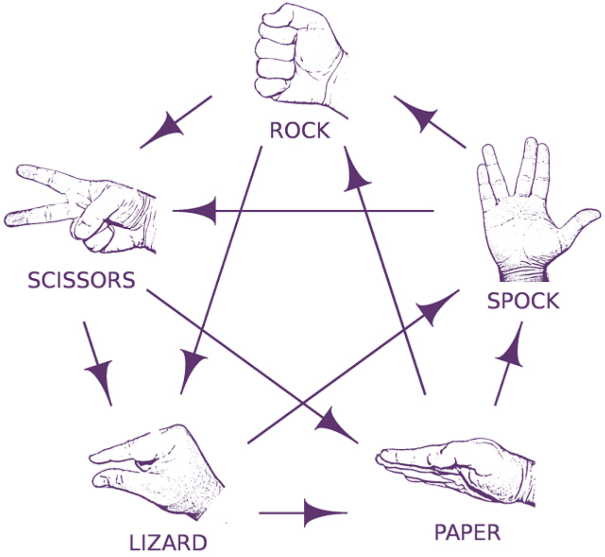
can just be

```
Scissors >> playAgainstPaper
  ^ #scissors
```

Remark


SPSElement
vs:
playAgainstScissors
playAgainStone
playAgainPaper
Stone
vs: anElement
playAgainstScissors
playAgainStone
playAgainPaper
Paper
vs: anElement
playAgainstScissors
playAgainStone
playAgainPaper
Scissors
vs: anElement
playAgainstScissors
playAgainStone
playAgainPaper

Extending it...



Conclusion

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

A course by

S. Ducasse, L. Fabresse, G. Polito, and Pablo Tesone