CHAPTER 5

Stone Paper Scissors

As we already saw sending a message is in fact making a choice. Indeed when we send a message, the method associated with the method in the class hierarchy of the receiver will be selected and executed.

Now we often have cases where we would like to select a method based on the receiver of the message and one argument. Again there is a simple solution named double dispatch that consists in sending another message to the argument hence making two choices one after the other.

This technique while simple can be challenging to grasp because programmers are so used to think that choices are made using explicit conditionals. In this chapter we will show an example of double dispatch via the paper stone scissors game.

5.1 Starting with a couple of tests

```
TestCase subclass: #StonePaperScissorsTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Stone new play: Paper new) = #paper
```

```
StonePaperScissorsTest >> testScissorIsWinning
  self assert: (Scissors new play: Paper new) = #scissors
```

```
StonePaperScissorsTest >> testStoneAgainsStone
  self assert: (Stone new play: Stone new) = #draw
```

5.2 Creating the classes

```
Object subclass: #Paper
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

```
Object subclass: #Scissors
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

```
Object subclass: #Stone
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'StonePaperScissors'
```

They could share a common superclass

5.3 With messages

```
StonePaperScissorsTest >> testPaperIsWinning
  self assert: (Stone new play: Paper new) = #paper
```

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

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

The test should pass now.

playAgainstStone:

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

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

Scissors now

```
StonePaperScissorsTest >> testScissorIsWinning
  self assert: (Scissors new play: Paper new) = #scissors
```

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

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

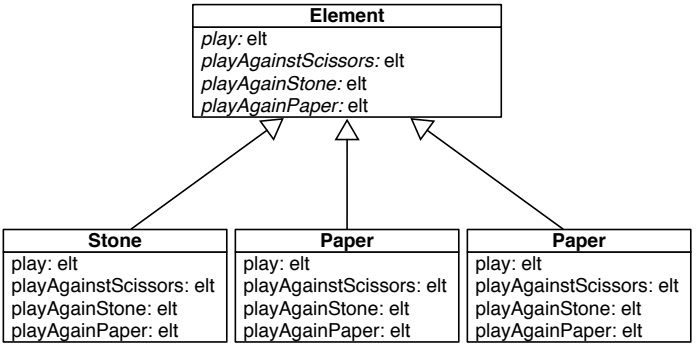

Figure 5-1 An overview of a possible solution using double dispatch.

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

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

Paper now

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

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

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

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

The methods could return a value such as 1 when the receiver wins, 0 when there is draw and -1 when the receiver loses. Add new tests and check this version.

5.4 A Better API

Both previous approaches either returning a symbol or a number are working but we can ask ourselves how the client will use this code.

Most of the time he will have to check again the returned result to perform some actions.

```
(aGameElement play: anotherGameElement) = 1
  ifTrue: [ do something for aGameElement]
  (aGameElement play: anotherGameElement) = -1
```

So all in all, while this was a good exercise to help you understand that we do not need to have explicit conditionals and that we can use message passing instead, it felt a bit disappointing.

But there is a much better solution using double dispatch. The idea is to pass the action to be executed to the object and that the object decide what to do.

```
Paper new competeWith: Paper new
  onDraw: [ Game incrementDraw ]
  onReceiverWin: [ ]
  onReceiverLose: [ ]
```

```
Paper new competeWith: Stone new
  onDraw: [ ]
  onReceiverWin: [ Game incrementPaper ]
  onReceiverLose: [ ]
```

Propose an implementation.

5.5 A possible implementation

```
Paper >> play: anElement onDraw: aDrawBlock onWin: aWinBlock onLose:
    aLoseBlock
  ^ anElement
    playAgainstPaper: self
    onDraw: aDrawBlock
    onReceiverWin: aWinBlock
    onReceiverLose: aLoseBlock
```

```
Paper >> playAgainstPaper: anElement onDraw: aDrawBlock
    onReceiverWin: aWinBlock onReceiverLose: aLoseBlock
  ^ aDrawBlock value
```

5.6 Conclusion

Sending a message is making a choice amongst several methods. Depending on the receiver of a message the correct method will be selected. Therefore sending a message is making a choice and the different classes represent the possible alternatives.

Now this example illustrates this point but going even further. Here we wanted to be able to make a choice depending on both an object and the argument of the message. The solution shows that it is enough to send back another message to the argument to perform a second selection that because of the first message now realizes a choice based on a message receiver and its argument.