CHAPTER 1

# 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 of 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 thinking that choices are made using explicit conditionals. In this chapter, we will show an example of double dispatch via the paper-stone-scissors game.

This exercise will show you an important paradigmatic shift where you will go from asking questions (conditionals) to sending orders. It is a clear illustration of the 'Don't ask, Tell' design principle.

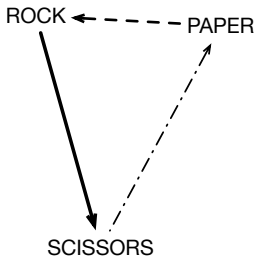


**Figure 1-1** Stone paper scissors.

## 1.1 Starting with a couple of tests

We start by implementing a couple of tests. Let us define a test class named `StonePaperScissorsTest`.

```
TestCase << #StonePaperScissorsTest
  package: 'StonePaperScissors'
```

Now we can define a couple of tests showing for example that a paper is winning when a stone plays against a paper. We consider that the following tests are self-explanatory.

```
StonePaperScissorsTest >> testStoneAgainstPaperIsWinning
  self assert: (Stone new play: Paper new) equals: #paper
```

```
StonePaperScissorsTest >> testScissorAgsinstPaperIsWinning
  self assert: (Scissors new play: Paper new) equals: #scissors
```

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

Define them because we will use the tests in the future.

## 1.2 Creating the classes

First, let us create the classes that will correspond to the different players.

```
Object << #Paper
  package: 'StonePaperScissors'
```

```
Object << #Scissors
  package: 'StonePaperScissors'
```

```
Object << #Stone
  package: 'StonePaperScissors'
```

They could share a common superclass but we left it to you.

## 1.3 With messages

We are ready to make sure that the first test is passing. Let us work on `testPaperIsWinning`.

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

The first method that we define is `play:` and it takes another player as an argument.

```
Stone >> play: anotherTool
  ... Your code ...
```

To implement this method we will use the fact that we know when its body is executed what is the receiver of the message. Here we are sure that the receiver is an instance of the class `Stone`.

So let us imagine that we have another method named `playAgainstStone:`

In the class `Paper`, it is clear that the method should return `#paper` because a paper wins against a stone. So just define it.

```
Paper >> playAgainstStone: aStone
  ... Your code ...
```

Now using the method `playAgainstStone:`, we can easily implement the previous method `play:` in the class `Stone`.

Do it and the test should pass now.

### playAgainstStone:

Since we have started to implement `playAgainstStone:`, let us continue and implement two other methods one in the class `Scissors` and the other in the class `Stone`.

In the class `Scissors` the method should return that a stone wins.

```
Scissors >> playAgainstStone: aStone
  ... Your code ...
```

In the class `Stone`, the method should return a draw.

```
Stone >> playAgainstStone: aStone
  ... Your code ...
```

Let us verify that the following tests are passing. For this, we only execute the tests whose receiver of the `play:` message are stone instance.

First, we add a test to check the new scenario and now we have all the scenarios where a stone is the receiver.

```
StonePaperScissorsTest >> testStoneAgainstScissorsIsWinning
  self assert: (Stone new play: Scissors new) equals: #stone
```

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

The case where a stone is the receiver of the message play is handled and we can pass to another class, for example, `Scissors`.

### Scissors now

Let us write first a test if this is already done. What we see is that a scissor is winning against a paper.

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

Now we are ready to define the corresponding methods. First, we define the methods `playAgainstScissors:` in the corresponding classes.

```
Scissors >> playAgainstScissors: aScissors
  ... Your code ...
```

```
Paper >> playAgainstScissors: aScissors
  ... Your code ...
```

```
Stone >> playAgainstScissors: aScissors
  ... Your code ...
```

Now we are ready to we define the method `play:` in the class `Scissors`.

```
Scissors >> play: anotherTool
  ... Your code ...
```

You can define a couple of tests to make sure that your code is correct.

```
StonePaperScissorsTest >> testScissorAgainstStoneIsLosing
  self assert: (Scissors new play: Stone new) equals: #stone
```

```
StonePaperScissorsTest >> testScissorAgainstScissors
  self assert: (Scissors new play: Scissors new) equals: #draw
```

### Paper now

We are now ready to do the same with the case of `Paper`. You should start to see the pattern. Define the method `playAgainstPaper:` in their corresponding classes.

```
Scissors >> playAgainstPaper: aPaper
  ... Your code ...
```

```
Paper >> playAgainstPaper: aPaper
  ... Your code ...
```

```
Stone >> playAgainstPaper: aPaper
  ... Your code ...
```

And now we can define the method `play:` in the `Paper` class.

```
Paper >> play: anotherTool
  ... Your code ...
```

Let us add more tests to cover the new cases.

```
StonePaperScissorsTest >> testPaperAgainstScissorIsLosing
  self assert: (Paper new play: Scissor new) equals: #scissors
```

```
StonePaperScissorsTest >> testPaperAgainstStoneIsWinning
  self assert: (Paper new play: Stone new) equals: #paper
```

```
StonePaperScissorsTest >> testPaperAgainstPaper
  self assert: (Paper new play: Paper new) equals: #draw
```

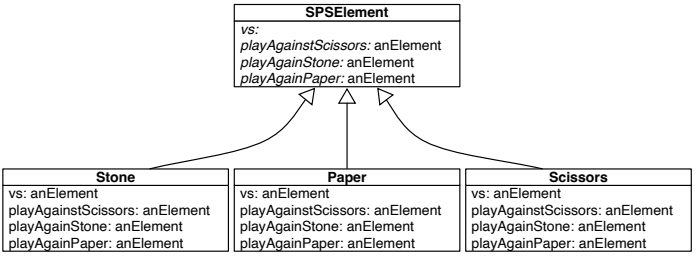


**Figure 1-2** An overview of a possible solution using double dispatch.

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

## 1.4 About double dispatch

This exercise about double dispatch is really simple and it has two aspects that you may not find in other situations:

First, it is symmetrical. You play a stone against a paper or the inverse. Not all the double dispatches are symmetrical. For example, when drawing an object against a canvas the operation for example `drawOn: aCanva` is directed. It does not change much about the double dispatch but we wanted to make clear that it does not have to be this way.

Second, the secondary methods (`playAgainstXXX`) do not use the argument and this is because the example is super simple. In real-life examples, the secondary methods do use the argument for example to call back behavior on the argument. We will see this with the visitor design pattern.

## 1.5 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 the object decides 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.

## 1.6 About alternative implementations

Here is a possible alternate 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
```

What we see is that this new API is not that nice. Being forced to create blocks is not that great. A possibility would be to pass an object that knows what to do.

```
Paper new competeWith: Paper new
  result: aResultHolder
```

Here is a sketch of a possible implementation:

```
Paper >> competeWith: anElement result: aResultHolder
  ^ anElement playAgainstPaper: self result: aResultHolder
```

We still have the double dispatch but we only need one object taking take of the results.

```
Stone >> playAgainstPaper: anElement result: aResultHolder
  aResultHolder paperWins
```

## 1.7 Conclusion

Sending a message is making a choice among 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 goes 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.