CHAPTER 2

Sending a message is making a choice

In this chapter we explore an *essential* point of object-oriented programming: Sending a message is making a choice! This aspect is often not really well put in perspective in teaching materials. Lectures often focus on inheritance but understanding the power of message passing it crucial to build good design. This point is so central for me that this is the first point that I explain when I start lectures on advanced design to people already understanding object-oriented programming.

I will start taking a simple example available in the core of Pharo: the Booleans. Pharo defines Booleans as two objects: `true` and `false`. They are so fundamental that you cannot change their value. I will explain how the Boolean negation and the disjunction (or) are implemented. Then I will step back and analyse the forces in presence and their importance.

2.1 Negation: the not message

Boolean negation has nothing special in Pharo: negating a boolean returned the negated value! For example the snippets below show this conventional behavior and vice versa.

Sending the message `not` to the Boolean `true` returns the Boolean `false`.

```
true not
>>> false
```

```
false not
>>> true
```

Nothing fancy. Of course the message `not` can be sent to Boolean expressions (i.e. expressions whose execution return Booleans) as shown below:

```
(2 * 4 > 3) not
>>> false
```

```
(#(1 2 3) includes: 5) not
>>> true
```

Now while Pharo follows traditional Boolean logic, what is less traditional is the implementation of the way the computation is done to answer the correct value.

2.2 Implementing not

Take a couple of minutes and a piece of paper and think about the way you would implement this message. Try really to write the code for real.

A first hint.

A first hint that I can give you is that the solution (used in Pharo and that we want to study) does not use explicit conditional such as `ifTrue:` or `ifTrue:ifFalse:`.

Take a bit more time to think how you can implement not. What we can tell you is the solution is not based on bit fiddling and logical operation on small integers. The solution we are looking for is simple and elegant.

A second hint.

The second hint is that `true` and `false` are instances of different classes. `true` is (the unique) instance of the class `True` while `false` is (the unique) instance of the class `False`. Note the uppercase on class names. This situation is depicted in Figure 2.1.

What you should see is that the fact that the solution has two different classes opens the door to have two different not implementations as shown by Figure 2.2. Indeed, as we mention in early chapters, we can have one message and multiple methods that we will be selected and executed depending on the receiver of the message.

Now you should be ready to get the solution. We should have a definition for the `true` defined in the class `True` and one for `false` in the class `False`.

Studying the implementation

The implementation of negation (message `not`) is defined as illustrated in Figure 2.3 and we shown below. The method `not` of the class `True` simply returns the Boolean `false`.

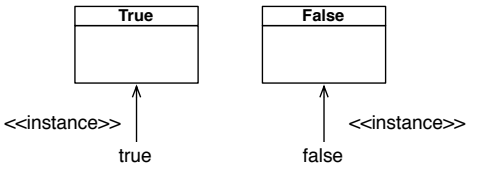

Figure 2.1 The two classes `True` and `False` and their respective unique instances `true` and `false`.

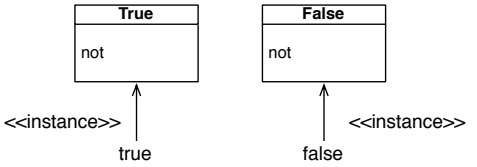

Figure 2.2 Two methods for one message.

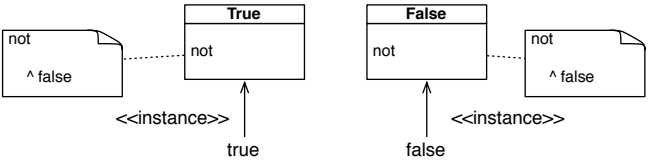

Figure 2.3 Two methods for one message each one returning the other instance.

```
True >> not
   "Negation--answer false since the receiver is true."
   ^ false
```

```
False >> not
   "Negation--answer true since the receiver is false."
   ^ true
```

Figure 2.4 shows that sending a message of one of the two Booleans selects the method in the corresponding class. What is important to see is that when a method is executed the receiver is from the class (or subclass we will see that later) that defines the method. We can also say that when we define a method in a given class we know that the receiver is from this class. Obvious, isn't it! But important. The implementation can then use this information as an execution context. This is exactly what the `not` implementation does. The method `not` defined on the class `True` knows that the receiver is true so it just has to return `false`.

Note When we define a method in a given class we know that the receiver is from this class. Obvious but important. The implementation can

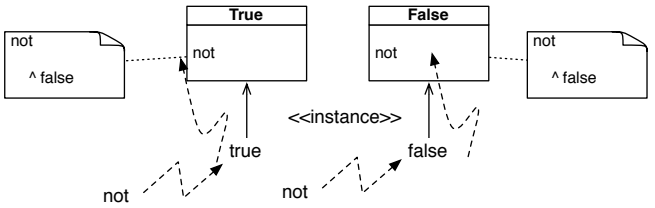

Figure 2.4 Sending a message selects the method in the class of the receiver .

- then use this information.

Now we will see if you get it... Let us try with a slightly more complex example.

2.3 Implementing disjunction

Disjunction is also a core functionality of any programming language. In Pharo disjunction is expressed via the message `|`. Here are the traditional tables describing disjunction but expressed in Pharo: first starting with `true` as receiver.

or	true	false
true	true	true
false	true	false

Here are a couple of examples expressed in Pharo.

```
true | true
>>> true
```

```
true | false
>>> true
```

```
false | false
>>> false
```

For the record, in fact the message `|` implements an eager disjunction since it asks the value of its argument even when not needed and Pharo also offers lazy disjunction implemented in the message `or:` which only requests the argument value if needed.

When receiver is true.

Propose an implementation of the disjunction for the first case: i.e. when the receiver is the object `true`.

or	true	false
true	true	true

What you should have learned from the implementation of `not` is that you have have two different methods taking advantage of the fact that they know what is the receiver during their execution.

```
true | true
>>> true
true | false
>>> true
true | anything
>>> true
```

When you look at the table we see that when the receiver is `true` the result is the same as the receiver (i.e. `true`). In Pharo the method `|` on class `True` express this as follows:

```
True >> | aBoolean
   "Evaluating Or -- answer true since the receiver is true."
   ^ true
```

When receiver is false.

Similarly let us study the Boolean table relative to false as receiver.

or	true	false
false	true	false

Here are some snippets

```
false | true
>>> true
false | false
>>> false
false | anything
>>> anything
```

We see that when the receiver is `false`, the result of the disjunction is the other argument. In Pharo the method `|` on class `False` is then as follows:

```
False >> | aBoolean
   "Evaluating Or -- answer with the argument, aBoolean."
   ^ aBoolean
```

2.4 About ifTrue:ifFalse: implementation

Now you should start to get the principle. Let us see how it works to also express conditional messages such as `ifTrue:ifFalse:`. Yes fundamental

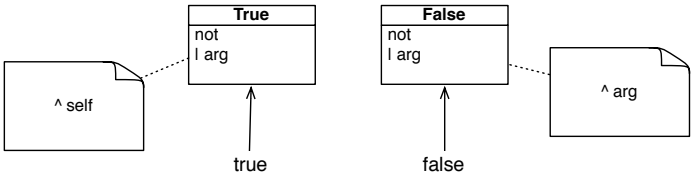

Figure 2.5 Disjunction implementation: two methods.

messages such as conditionals can be expressed using the same mechanism: late binding.

What you see with the following snippet is that message `ifTrue:ifFalse:` is expecting two different blocks as argument.

```
4 factorial > 20
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
>>> 'bigger'
```

Now you should know that execute a block you should use the message `value` as illustrated:

```
[1 + 3] value
>>> 4
```

Block can contain any expressions. The execution of the following block will open the Pharo logo.

```
[ (ZnEasy getPng: 'http://pharo.org/web/files/pharo.png')
       asMorph openInWindow ] value
```

Based on the receiver we should execute the corresponding block. When the expression (`4 factorial > 20` in the example above) is true we should execute the `ifTrue:` argument, when it is false we should execute the `ifFalse:` argument.

Implementation.

The implementations is then simple and elegant. In the `True` class, we want to execute the corresponding block, the one passed as `ifTrue:` argument as shown in Figure 2.6.

```
True >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
   ^ trueAlternativeBlock value
```

Similarly in the `False` class, we want to execute the corresponding block, the one passed as `ifFalse:` argument.

```
False >> ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
   ^ falseAlternativeBlock value
```

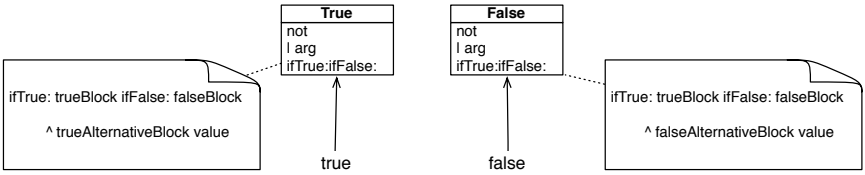

Figure 2.6 Conditional implementation: again two methods and no explicit tests.

Optimisation.

What we show above works! But if you modify it, the modification will not be taken into account. This is because in Pharo `ifTrue:ifFalse:` is used so often and its semantics should not change that the compiler in fact does not send a message but convert it in low-level logic for the virtual machine. Now you can invent your own conditional message `siVrai:siFaux:` for a french version for example and you will see that this implementation works.

2.5 What is the point?

Some readers may be intrigued and think that this is spurious because they will never have to reimplement Booleans in their life. This is true even if there are different versions of Boolean logic such as the ternary logic that contains also unknown value.

Now we picked this example to illustrate an important point: sending a message is making a choice. The runtime system will dynamically select the method depending on the receiver. This is what is called late binding or dynamic dispatch. Only at execution the correct method is selected. Now the Boolean example is the simplest one I could find to illustrate this point. It is also ground breaking in the sense that it touches something as fundemantal as Boolean main operations.

Now the choices can be made over multiple tenth of classes. For example in Pillar the document processing system in which this book is written there are around 35 different classes expressing different parts of a document: section, title, bold, paragraph... and the same principle applies there. The system selects the correct methods to render text, LaTeX or HTML using exactly the same principle.

Now most of the time you can express the same using conditions (except for the Boolean example and this is why I took this example).

```
emitHTML: stream
  self == Title
    ifTrue: [ ... ]
    self == Section
```

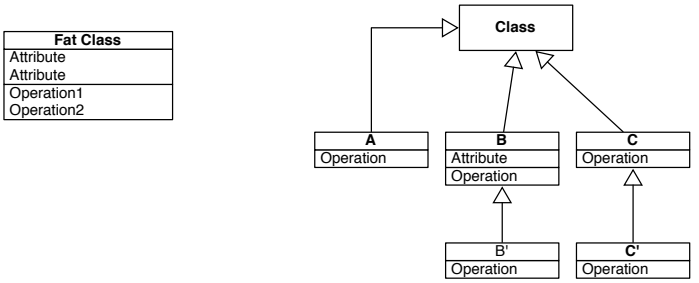

Figure 2.7 One single class vs. a nice hierarchy.

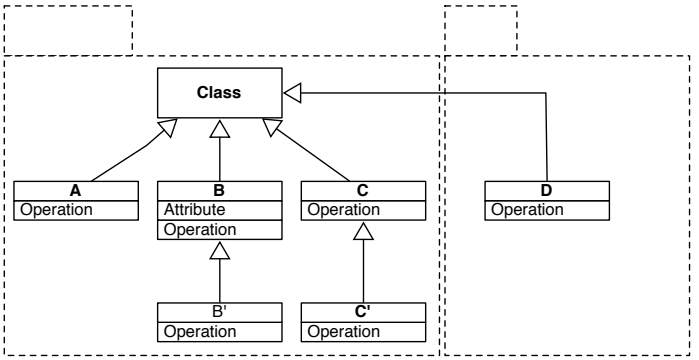

Figure 2.8 One single class vs. a nice hierarchy.

```
    ifTrue: [ ... ]
    ...
```

The problems with such explicit conditions is the following:

- First, there are cumbersome to write. Even using case statements as in other language the logic can become complex. Imagine for 35 cases of Pillar.
- Second, such definitions are not modular. It means that adding a new case requires to edit the method and recompile it. While with the dynamic dispatch, we can just add a new class as shown in Figure 2.7. Furthermore this class can just take advantage of an existing one and extend it (as we will in the Chapter **??**).

You could think that this is a not a problem but imagine that now for a business you want to ship different products or solutions to your clients. With dynamic dispatch you can simply package alternate code in separate packages and load them independently as shown in Figure **??**.

Classes represent choices

Sending a message is making a choice. Now the following question is what elements represent choices. Because you can have the possibility to chose something but if there is only one choice you will not go that far and take advantage of the power of late binding.

In fact classes represent choices. In the Boolean case you have two choices one for true and one for false. There is a really difference for example between the FatClass design (left in Figure2.7) and the modular design (right in Figure2.7) because we see all the choices that can be made at runtime in the latter case.

When I do code review, I looked at how domain variations are represented and if there are enough classes. What is important to realise is that classes are cheap. It is better to write 5 little classes than a huge one. Some (even smart) people got confused by measuring complexity of a system using number of classes. Having many classes representing good abstractions with a single responsibility is much much better than having a single class exhibiting multiple responsibilities.

2.6 Conclusion

Sending a message is really powerful since it selects the adequate method to be executed on the receiver. In future chapters we will show that sending a message is creating in fact a hook so that other methods can be executed in place.

Now you should start to understand why in Pharo we are picky about the vocabulary: we use sending a message and not calling a method as in other language. Because sending a message conveys much better the idea that the correct method will be selected and that we do not know a priori which one will be executed.

Remember that when we execute a method (and also write it), one key information we get is that the receiver from this class or one of its subclasses as we will later.