Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Objects and classes

Pharo is a pure object-oriented programming language, i.e., everything in the system is an object i.e., an entity created by a class and reacting to messages.

This chapter presents key mechanisms that characterize object-oriented programming: *objects*, *classes*, *messages* and *methods*. We will also present *distribution of responsibilities* which is one of the heart of object-oriented programming as well as *delegation* and *composition*. Each of these mechanisms will be used and illustrated again in this book.

We start explaining objects, classes, messages and methods with really simple examples. Then in the following chapter we will propose an example that illustrates what we can achieve by using objects of different classes.

Objects are created by *classes* that are object factories: Classes define the structure and behavior of objects (in terms of methods) but each object has a specific state and identity that is unique and different from all other objects. A class defines *methods* that specify how a *message* is actually implemented.

1.1 Objects: Entities reacting to messages

Instead of a bit-grinding processor ... plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires. [Ingall 81]

Object-oriented programming is about creating objects and interacting with objects by sending them *messages*.

Objects are entities that communicate via messages and react to messages by executing certain tasks. Moreover objects hide the way they define these

tasks: the client of an object send a message to an object and the system find the corresponding method to be executed. Messages specify what should be done and methods how it should be done.

Turtles as an example

Imagine that we have a graphics turtle like a LOGO turtle. We do the following: create a turtle, send it messages to make it move, turn, and trace some drawings. Let us look at this in detail.

Creating an object

First we create a new turtle by sending the message `new` to the class `Turtle`.

```
| t |
t := Turtle new.
```

A class is a cast for objects. All the objects, instances of a class, share the same characteristics and behavior. For example, all the turtle instances have a direction and understand messages to rotate and move. However, each turtle has its own value for its direction. We say that all the instances of a class have the same instance variables but each as private value for them.

Sending messages

The only way to interact with objects is to send them *messages*. In the following snippets we send messages

- to create an object , message `new`,
- to tell the turtle to turn, message `turn:`, and
- to tell the turtle to move, message `go:`.

```
| t |
t := Turtle new.
t turn: 90.
t go: 100.
t turn: 180.
t go: 100.
```

When an object receives a message, it reacts by performing some actions. An object can return a value, change its internal state, or send messages to other objects. Here the turtle will change its direction and it will interact with the display to leave a trail.

Multiple instances: each with its own state.

We can have multiple objects of the same class and each one has a specific state. Here we have two turles each one located to a specific position and pointing into its own direction.

```
| t1 t2 |
t1 := Turtle new.
t1 turn: 90.
t1 go: 100.
t1 turn: 180.
t1 go: 100.
t2 := Turtle new.
t2 go: 100.
t2 turn: 40.
t2 go: 100.
```

1.2 Messages and Methods

Messages specify *what* the object should do and not how it should do it (this is the duties of methods). When we send the message `go:` we just specify what we expect the receiver to do. Sending a message is similar to the abstraction provided by procedures or functions in procedural or functional programming language: it hides implementation details. However sending a message is much more than executing a sequence of instructions: it means that we have to find the method that should be executed in reaction to the message.

Message: what should be executed

The message `square:` is send to a new turtle with 100 as argument. The message expresses what the receiver should do.

```
Turtle new square: 100
```

Method: how we execute it

The method definition `square:` below defines step by step what are the actions to be done in response to the message `square:`. It defines that to draw a square the turtle receiving the message `square:` (represented by `self`) should perform four times the following sequences of messages: move forward a distance (message `go:`), turn 90 degrees (using the message `turn:`).

```
square: size
  4 timesRepeat: [ self go: size; turn: 90 ]
```

Note that finding the method corresponding to the message is done at runtime and depends on the object receiving the message.

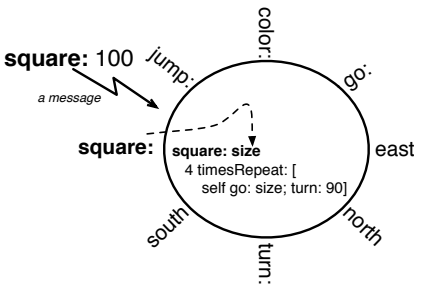

Figure 1-1 An object presents to the other objects an interface composed of a set of messages defining *what* he can do. This behavior is realized by methods that specify *how* the behavior is implemented. When a message is sent to an object a method with the message name (called selector) is looked up and executed.

- A message represents *what* the object should do, while a method specifies *how* the behavior is realized.

  An object can also send messages to other objects. For example, when a turtle draws a line, it sends messages to an object representing the line color and its length.

- An object is an entity that once created receives messages and performs some actions in reaction. When a message is sent to an object, a method with the message name is looked up and executed.

1.3 An object is a protective entity

An object is responsible of the way it realizes its reaction to a message. It *offers services* but *hides* the way they are implemented (see Figure 1-2). We do not have to know how the method associated with the message selector is implemented. Only the object knows the exact definition of the method. This is when we define the method `square:` that defines how a turtle draws a square of a given size, that we focus on *how* a turtle draws a square. Figure 1-2 shows the message and the method `square:`. The method `square:` defines how to draw step by step a square, however the object only offers the message square: and does not show it implementation.

> **Important** An object presents to the other objects an *interface* (i.e., a set of messages) defining *what* the object can do. This behavior is realized by methods that specify *how* the behavior is implemented. To perform something useful some data are most of the time required. Data are only accessed by the methods.

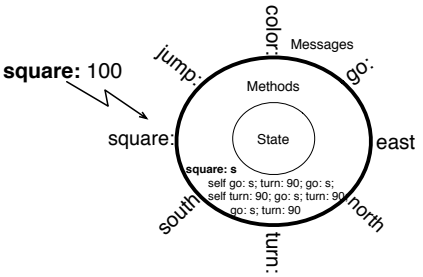

Figure 1-2 The message `square:` can be implemented differently. This different implementation does not impact the sender of the message who is not concerned by the internals of the object.

From a turtle *user* point of view, the only relevant information is that the turtle effectively receiving the message `square:` executes the method that draws a square. So changing the definition of the `square:` method to the one below does not have any consequence on the methods that call it. Figure 1-2 illustrates this point.

```
square: s
   "Make the receiver draw a square of size s"

   self go: s; turn: 90; go: s; turn: 90.
   self go: s; turn: 90; go: s; turn: 90
```

Hiding the internal representation is not limited to object-oriented programming but it is central to object-oriented programming.

Important An object is responsible of the way it realizes its reaction to a message. It offers services and hides the way they are implemented.

1.4 An object protects its data

An object holds some *private data* that represents its state (see Figure 1-3). Moreover, it controls its state and should not let other objects play directly with them because this could let him into an inconsistent state. For example, you do not want to somebody else plays with the data of your bank account directly and really want to control your transaction.

For example, a LOGO turtle can be represented by a position, a direction and a way to indicate if its pen is up or down. But, we cannot directly access these data and change them. For that we have to use the set of messages proposed by a turtle. These methods constitute the *interface* of an object. We say that the object state is *encapsulated*, this means that not everybody can access

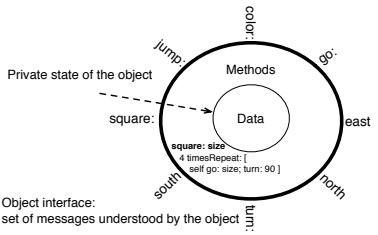

Figure 1-3 A turtle is an object which has an interface, i.e., a set of messages to which it can reply and a private state that only its methods can access.

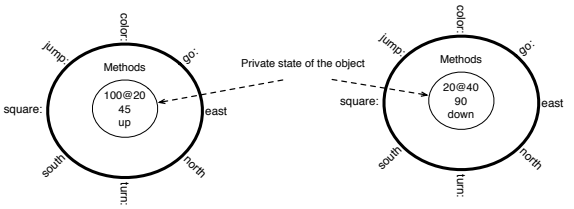

Figure 1-4 Two turtles have the same interface, i.e., set of messages being understood but they have *different* private state representing their direction, position and pen status.

it. In fact, object-oriented programming is based on encapsulation, i.e., the fact that per default objects are the only ones that can access their own state.

In Pharo, a client cannot access the state of an object if the object does not define a method to access it. Moreover, clients should not rely on the internal representation of an object because an object is free to change the way it implements its behavior. Exposing the internal state of an object by defining methods providing access to the object data weakens the control that an object has over its own state.

> **Important** An object holds some *private* data that represents its *internal* state. Each object has its own state. Two objects of the same class share the same *interface* but have their own private state.

1.5 With counters

Now that you got the main point of objects, we can see that it applies to everything. In Pharo *everything* is an object. In fact there is *nothing* else, only objects. Here is a little program with counters.

We create two counters that we store in variables `c1` and `c2` instances of the class `Counter`. Each counter has its own state but exhibits the same behavior as all the counters defined by the class `Counter`:

- when responding to the message `count`, it returns its value,
- when responding to the message `increment`, it increment one to its current value.

```
| c1 c2 |
c1 := Counter new.
c2 := Counter new.
c1 count.
>>> 0
c1 increment.
c1 increment.
c1 count.
>>> 2
c2 count.
>>> 0
c 2 increment.
c2 count.
>>> 1
```

1.6 A class: blueprint or factory of objects

A class is a mold or cast of objects. A class specifies two important aspects of their instances:

- Instance **structure**. All the instances of a class will have the same structure expressed in terms of *instance variables*. Pay attention that the variables are the same for all the instances of a class but not their values. Each instance has specific values for its instance variables.
- Instance **behavior**. All the instances share the same behavior even if this one can be different because applied on different values.

> **Important** A class is as a blueprint for its instances. It is a factory of objects. All objects will have the same structure and share a common behavior.

Let us illustrate this with the class `Counter`.

Object structure

Let us study the `Counter` class definition.

```
Object subclass: #Counter
  instanceVariableNames: 'count'
  classVariableNames: ''
  package: 'LOOP'
```

The expresion `Object subclass: #Counter` indicates that the class `Counter` is a subclass of the class `Object`. It means that counter instances understand the messages defined also by the class `Object`. In Pharo, classes should at least be a subclass of the class `Object`. You will learn more about subclassing and inheritance in Chapter **??**.

Then the class `Counter` defines that all the instances will have one instance variable named `count` using the expression `instanceVariableNames: 'count'`. And each instance of the class `Counter` will have a `count` variable with a *different* value as we showed in the examples above.

Finally the class is defined in the package `'LOOP'`. A package is a kind of folder containing multiple classes.

Object behavior

In addition a class is the place that groups the behavior of its instances. Indeed since all the instances of the class share the *same* behavior definitions, such behavior is defined and grouped in a class.

For counters, the class defines how to retrieve the counter value, how to increment and decrement the count as used in the messages in the previous code snippets.

Here is the definition of the **method** `increment`. It simply adds one to the instance variable `count`.

```
Counter >> increment
  count := count + 1
```

When we send a message to a counter for example in the expression `c1 increment`, the method `increment` will be applied on *that* specific object `c1`. In the expression `c1 increment`, `c1` is called the **receiver** of the message `increment`.

In the method `increment`, the variable `count` refers to the variable of the **receiver** of the message.

A class defines methods that specify the behavior of all the instances created by the class.

Multiple methods can accessed to the instance variables of the receiver. For example the methods `increment`, `count: decrement` and `printOn:` all access the instance variable `count` of the receiver to perform different computation.

```
Counter >> count: anInteger
  count := anInteger
```

```
Counter >> decrement
  count := count - 1
```

```
Counter >> printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: ' with value: ', self count printString.
```

For example, once the following program is executed the count instance variable of the counter `c2` will hold the value 11, since the method `count:` will set its value to 10, and `increment` will set it to 11 and 12 and finally `decrement` will set it to 11.

```
| c2 |
c2 := Counter new.
c2 count: 10.
c2 increment.
c2 increment.
c2 decrement.
```

Self is the message receiver

Imagine that now we would like to send a message to the object that receives the message itself. We need a way to refer to this object. Pharo defines a special variable for this exact purpose: `self`.

Important `self` always refers to the message receiver that is currently executed.

For example we can implement the method `incrementByTwo` as follows:

```
Counter >> incrementByTwo
  self increment.
  self increment
```

When we execute the expression `c1 incrementByTwo`, during the execution of the method `incrementByTwo`, `self` refers to `c1`.

We will explain how a method is found when a message is sent but first we should explain inheritance, i.e., how a class is defined incrementally from a root class and all this will be explained in Chapter **??**.

1.7 Class and instances are really different

Classes and objects are different objects; they understand different messages.

For example, sending `new` to the `Counter` class returns a newly created counter, while sending `new` to a counter results in an error. In the opposite way, send-

ing `increment` to the class `Counter` leads also to an error because the class `Counter` is a factory of objects not the objects themselves.

A class is a factory of objects. A class creates instances. An instance does not create other instances of the class.

A class describes the structure (instance variables) and the behavior (methods) of *all* its instances. The state of an instance is the value of its instance variables and it is specific to one single object while the behavior is shared by all the instances of a class.

1.8 Conclusion

In this chapter you saw that:

- An object is a computer entity that once created receives messages and performs some actions in reaction.
- An object has an unique identity.
- An object holds some private data that represent its internal state.
- A class is a factory of objects: It *describes* the internal structure of all its instances by means of instance variable.
- All objects of the same class share the same behavior, i.e., the same method definitions.
- Instance variables are accessible by all the methods of a class. Instance variables have the same lifetime than the object to which they belong to.
- In Pharo , instance variables cannot be accessed from outside of an object. Instance variables are only accessible from the methods of the class that define them.
- Methods define the behavior of all the instances of the class they belong to.

CHAPTER 2

Revisiting objects and classes

In the previous chapter we presented objects and classes via simple examples. In this chapter we introduce a little bit more elaborated example: a little file system where we revisit everything and extend it to explain *late binding*, *distribution of responsibilities* and *delegation*. The file example will be extended to present *inheritance* in Chapter **??**.

2.1 A simple and naive file system

We start to present a simple example that we use to present and explain the concepts: a simple and naive file system as shown in Figure 2-1. What the diagram shows is that we have:

- files that also have a name and a contents. Here we get three different files `Babar`, `Astroboy` and `tintinEtLesPicaros`.
- directories that have a name and can contain other files or directories. Here we get the `manga`, `comics`, `oldcomics` and `belgiumSchool` directories. Directories can be nested: `comics` contains three repositories. The `belgiumSchool` directory contains `tintinEtLesPicaros`.

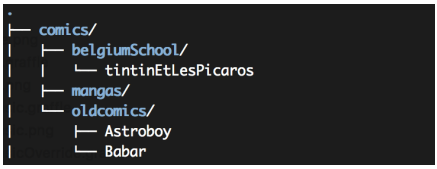

Figure 2-1 Some directories and files organised in a file system.

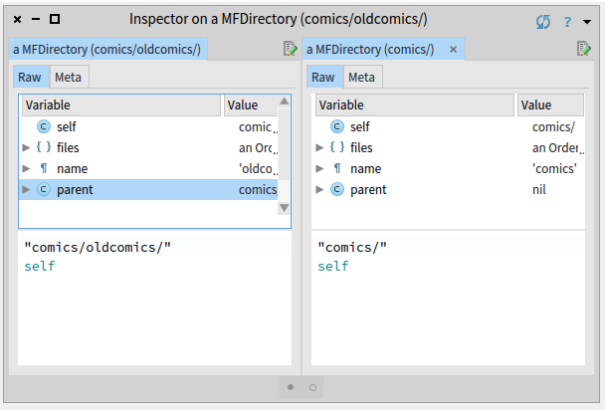

Figure 2-2 Inspecting `dOldComics` and clicking on the `parent` variable.

2.2 Studying a first scenario

Since what we want to develop may be a bit unclear for us, let us define first an example. In the rest of this book we will code such examples as tests that can automatically be executed. For now it would make the discourse too complex, so we just use little code examples.

We create two directories.

```
| dComics dOldComics dManga |
dComics := MFDirectory new name: 'comics'.
dOldComics := MFDirectory new name: 'oldcomics'.
```

We add the oldcomics folder to comics and we check that the parent children relationship is well set.

```
  ...
dComics addElement: dOldComics.
dOldComics parent == dComics
>>> true
```

Here we verify that the parent of `dOldComics` is `dComics`: the message `==` checks that the receiver is the same object than the argument.

You can also inspect the receiver as follows and if you click on the instance variable parent of the receiver you should obtain the situation depicted by Figure 2-2.

```
  ...
dOldComics inspect
```

We continue with some queries.

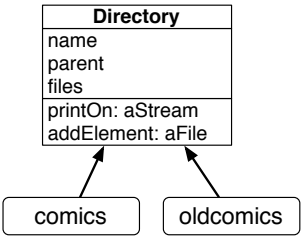

Figure 2-3 The `Directory` class and some instances (directories).

```
  ...
dComics parent
>>> nil
```

Here we verify that `dOldComics` is comprised in the children of `dComics`.

```
  ...
dComics children includes: dOldComics.
>>> true
```

We create a new repository and we check that once added to a parent repository, it is included in the children.

```
dManga := MFDirectory new name: 'manga'.
dComics addElement: dManga.
dComics children includes: dManga
>>> true
```

2.3 Defining a class

Let us start by defining the directory class.

```
Object subclass: #MFDirectory
  instanceVariableNames: 'parent name files'
  classVariableNames: ''
  package: 'MyFS'
```

When we create a directory, its files is an empty ordered collection. This is what we express in the following method `initialize`.

```
MFDirectory >> initialize
  files := OrderedCollection new
```

A newly created object is sent the message `initialize` just after its creation. Therefore the `initialize` method is executed.

Now we can write the method `addElement:`. (To keep things simple, note that we consider that when a file is added to a directory, it was not belonging to a another directory. This behavior could be implemented by `aFile`

`moveTo: aDirectory`) Adding a file to a directory means: (1) that the parent of the file is changed to be the directory to which it is added, (2) that the added file is added to the list of files contained in the directory.

```
MFDirectory >> addElement: aFile
  aFile parent: self.
  files add: aFile
```

Note that the method name `addElement:` is not nice but we chose it on purpose so that you do not believe that delegating requires that the methods have the same name. An object can delegate its part of duties to another object by simply passing a message.

We should then define the methods `name:`, `parent:`, `parent`, and `children` to be able to run our example.

```
MFDirectory >> name: aString
  name := aString
```

```
MFDirectory >> parent: aFile
  parent := aFile
```

```
MFDirectory >> parent
  ^ parent
```

```
MFDirectory >> children
  ^ files
```

With such method definitions, our little example should run. It should not print the same results because we did not change the printing of the objects yet.

A first little analysis

When we look at the implementation of the method to add a file to a directory we see that the class `MFDirectory` used another class `OrderedCollection` to store the information about the files it contains. An ordered collection is a quite complex object: it can insert, remove elements, grow its size, and many more operations.

We say that the class `MFDirectory` delegates a part of its duties (to keep the information of the files it contains) to the class `OrderedCollection`. In addition, when an object is executed, the object to which it may delegate part of its computation may change dynamically.

Such behavior is not specific to object-oriented programming, in procedural languages we can call another function defined on a data structure. Now with object-oriented programming, there is a really important point: an object will send messages to other objects (even from the same class) and such message send will use the message offered by the receiver. There is normally no way for an object to access the internal structure of another object.

2.4 Printing a directory

Now we would like to get the directory printed in a better way. Without too much explanation, you should know that the method `printOn: astream` of an object is executed when the system or we send the message `printString` to an object. So we can specialise it.

The argument passed to the method `printOn:` is a stream. A stream is an object in which we can store information one after the other in sequence using the message <<. The argument of << should be a sequence of objects such as string (which is a sequence of characters).

```
MFDirectory >> printOn: aStream
  aStream << name
```

Let us try.

```
| el1 el2 |
el1 := MFDirectory new name: 'comics'.
el2 := MFDirectory new name: 'oldcomics'.
el1 addElement: el2.
el1 printString
>>> 'comics'
```

```
  ...
el2 printString
>>> 'oldcomics'
```

What would be nice is to get the full path so that we can immediately understand the configuration. For example we would like to finish with a '/' to indicate that this is a directory as with the ls command on unix.

```
| el1 el2 |
el1 := MFDirectory new name: 'comics'.
el2 := MFDirectory new name: 'oldcomics'.
el1 addElement: el2.
el1 >> printString.
>>>'comics'
```

```
  ...
el2 printString
>>> 'comics/oldcomics/'
```

A possible definition is the following one:

```
MFDirectory >> printOn: aStream
  parent isNil
    ifFalse: [ parent printOn: aStream ].
  aStream << name.
  aStream << '/'
```

Try it and it should print the expected results. What do we see with this definition: it is a kind of recursive definition. The name of a directory is in fact

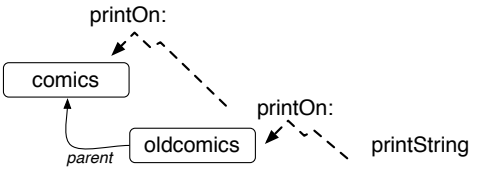

Figure 2-4 Navigating an object graph by sending message to different objects.

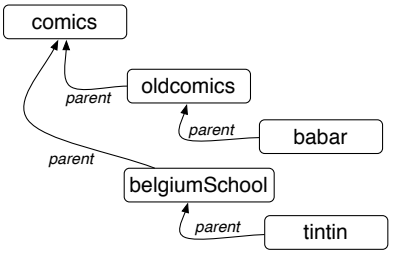

Figure 2-5 A graph of objects to represent our file system.

the concatenation (here we just add in the stream but this is the same.) of the name of its parents (as shown in Figure 2-4). Similar to a recursive function navigating a structure composed of similar elements (like a linked-list or any structure defined by induction), each parent receives and executes another time the `printOn:` method and returns the name for its part.

2.5 Adding files

Now we want to add files. Once we will have defined files we will be able to have a graph of objects of different kinds represent our file system with directories and files as shown in Figure 2-5.

An example first

Again let us start with an example. A file should contain some contents.

```
| el1 dOldComics |
el1 := MFFile new name: 'astroboy'; contents: 'The story of a boy
    turned into a robot that saved the world'.
dOldComics := MFDirectory new name: 'oldcomics'.
dOldComics addElement: el1.
el1 printString.
>>>
'oldcomics/astroboy'
```

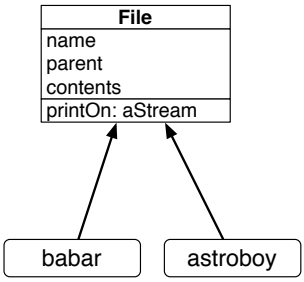

Figure 2-6 A new class and its instances.

A new class definition

Again a file needs a name, a parent and in addition a contents.

We define the class `MFFile` as follows and illustrated in Figure 2-6. Note that this solution is not satisfactory and we will propose a much better one later.

```
Object subclass: #MFFile
  instanceVariableNames: 'parent name contents'
  classVariableNames: ''
  package: 'MyFS'
```

As for the directories we initialize the contents of a file with a default value.

```
MFFile >> initialize
  contents := ''
```

We should define the same methods for `parent:`, `parent` and `name:`. This duplication coupled with the fact that we get nearly the same class definition should be a clear warning. It means that we do not reuse enough and that if we want to change the system we will have to change it multiple times and we may introduce errors by forgetting one place. We will address it in Chapter **??**. In addition we will add a method to be able to set the contents of the file `contents:`.

```
MFFile >> name: aString
  name := aString
```

```
MFFile >> parent: aFile
  parent := aFile
```

```
MFFile >> parent
  ^ parent
```

```
MFFile >> contents: aString
  contents := aString
```

At the stage we should be able to define a file and adding it to a directory.

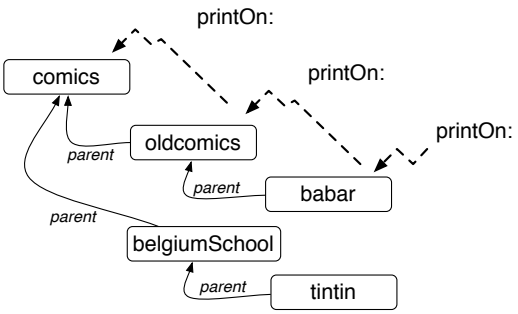

Figure 2-7 Printing a file: Sending messages inside a graph of different objects.

Now we should redefine the implementation of `printOn:` to print nicely the name of file:

```
MFFile >> printOn: aStream
  aStream << name
```

But this is not enough because we will just get `'astroboy'` and not `'oldcomics/astroboy'`. So let us improve it.

```
MFFile >> printOn: aStream
  parent isNil ifFalse: [
    parent printOn: aStream ].
  aStream << name
```

2.6 One message and multiple methods

Before continuing let us step back and analyse the situation. We send the same messages and we execute different methods.

```
| el1 dOldComics dComics |
el1 := MFFile new name: 'astroboy'; contents: 'The story of a boy
    turned into a robot that saved the world'.
dOldComics := MFDirectory new name: 'oldcomics'.
dComics := MFDirectory new name: 'comics'.
dComics addElement: dOldComics.
dOldComics addElement: el1.
el1 printString.
>>>
'comics/oldcomics/astroboy'
```

```
dOldComics printString.
>>>
'comics/oldcomics/'
```

What we see is that there is one message and several implementations of methods and that sending a message will find and execute the correct method.

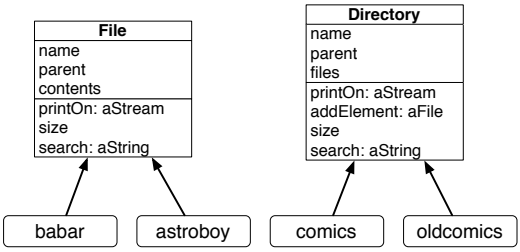

Figure 2-8 Two classes understanding similar sets of message.

For example, there are two methods `printOn:` one for file and one for directory but only one message `printOn:` sent from the `printString` message.

In addition a method can be defined in terms of messages sent to other objects. The method `printOn:` for directories is complex and it delegates the same message to other objects, its parents (as illustrated by Figure 2-7). The method `addElement:` delegates to the OrderedCollection sending a different message `add:`.

2.7 Objects: stepping back

Now that we saw some examples of objects, it is time to step back. Objects are defined by the values of their state, their behavior (shared with the other instances of their class) and an identity.

- **State.** Each object has specific values. While all the instances of classes have the same structure, each instance has its own values. Each object has a private state. Clients or users of an object cannot access the state of the object if this one does not explicitly expose it by defining a method returning it (such as the message `count`).
- **Behavior.** Each object shares the same behavior with all the instances of its class.
- **Identity.** An object has an identity. It is unique. `oldcomics` is clearly not the same as `comics`.

2.8 Examples of distribution of responsibilities

We will now implement two functionalities: the size of directories and a search based on the contents of the files. This will set the context to explain the key concept of distribution of responsibilities.

File size

Let us imagine that we want to compute the size of a directory. Note that the size computation we propose is fantasist but this is for the sake of the example. To perform such a computation we should also define what is the size of a file. Again let us start with examples (that you will turn into tests in the future.).

First we define the file size as the size of its name plus the size of its contents.

```
| el |
el := MFFile new name: 'babar'; contents: 'Babar et Celeste'.
el size = 'babar' size + 'Babar et Celeste' size.
>>> true
```

Second we define the directory size as its name size plus the size of its files and we add and arbitrary number: 2.

```
| p2 el |
el := MFFile new name: 'babar'.
p2 := MFDirectory new name: 'oldcomics'.
p2 addElement: el.
p2 size = 'oldcomics' size + 'babar' size + 2
>>> true
```

We define two methods `size` one for each class (see Figure 2-8).

```
MFFile >> size
  ^ contents size + name size
```

```
MFDirectory >> size
  | sum |
  sum := 0.
  files do: [ :each | sum := sum + each size ].
  sum := sum + name size.
  sum := sum + 2.
  ^ sum
```

Search

Let us imagine that we want to search the files matching a given string. Here is an example to set the stage.

```
| p el1 el2 |
p := MFDirectory new name: 'comics'.
el1 := MFFile new name: 'babar'; contents: 'Babar et Celeste'.
p addElement: el1.
el2 := MFFile new name: 'astroboy'; contents: 'super cool robot'.
p addElement: el2.
(p search: 'Ba') includes: el1
>>> true
```

To implement this behavior is quite simple: we define two methods one in each class (as shown in Figure 2-8).

```
MFFile >> search: aString
  ^ '*', aString, '*' match: contents
```

```
MFDirectory >> search: aString
  ^ files select: [ :eachFile | eachFile search: aString ]
```

2.9 Important points

These two examples show several *important* points:

Modular thinking

Each method is modular in the sense that it only focuses on the behavior of the objects specified by the class defining the method. Such method can be built by sending other messages without having to know how such methods are defined. It also means that we can add a new kind of classes or remove one without having to change the entire system.

Sending a message is making a choice

We send *one* message and one method amongst the *multiple* methods with the same name will be selected and executed. The method is dynamically looked up during execution as we will see in Chapters **??** and **??**. Sending a message is selecting the corresponding method having the same name than the message. When a message is sent to an object the corresponding method is looked in the class of the message receiver.

> **Important** Sending a message is making a choice. The system selects for us the correct method to be executed.

Polymorphic objects

We created objects (files and directories) that are *polymorphic* in the sense that they offer a common set of messages (`search:`, `printOn:`, `size`, `parent:`). This is really powerful because we can compose objects (for example add a new directory or a file) without changing the program. Imagine that we add a new kind of directories we can introduce it and reuse extending programs based on `size` or `search:` *without* changing them.

> **Important** Creating polymorphic objects is a really powerful capability. It lets us extend and change programs without breaking them.

Most of the time it is better to give similar name to methods performing similar behavior, and different names when the methods are doing semantically different actions, so that users of the objects are not confused.

The polymorphism is really a strength of object-oriented languages because it allows one to treat different objects, i.e., instances of different classes, uniformly as soon as they implement the same messages. Polymorphism works in synergy with the idea that an object is responsible to decide how to react to message reception. Indeed, the fact that different objects can implement the same messages let us write code that only tell the objects to execute some actions without worrying exactly about the kind of objects.

2.10 Distribution of responsibilities

This example as well as the printing of files and directories illustrates something fundamental in object-oriented programming: the distribution of responsibilities. With the distribution of responsibilities, each kind of objects is responsible for a specific behavior and a more elaborated behavior is composed out of such different behavior. The size of a directory is computed based on the size of its files by requesting the files to compute their size.

Procedural

Let us take some time to compare with procedural thinking. Computing the size of a list of files and directories would have been expressed as a monolitic behavior sketch below:

```
sizeOfFiles: files
  | sum |
  sum := 0.
  files do: [ :aFile |
    aFile class = MFFile
      ifTrue: [ sum := sum + aFile name size + aFile contents size ].
    aFile class = MFDirectory
      ifTrue: [
        | fileSum |
        fileSum := 0.
        each files do: [:anInsideFile | fileSum := fileSum +
    anInsideFile name size + anInsideFile contents size ].
        sum := sum + fileSum + each name size + 2].
  ^ sum
```

While this example is a bit exagerated, we see several points:

- First, we explicitly check the kind of structures we are manipulating. If this is a file or directory we do something different.

- Second, the logic of the computation is defined inside the `sizeOfFiles:` itself, and not in the entities themselves. This means in particular that such logic cannot be reused.
- A part of the implementation logic is exposed and not in control of the object. It means that if we decide to change the internal structure of our classes, we will have to change this function too.
- Adding a new kind of such as a root directory is not modular. We will have to modify the method `sizeOfFiles:` function.

What you should also see when you compare the two versions is that in the procedural version we have to check the kind of object we manipulate. In the object-oriented version, we simply tell the object to perform its own computation and return the result to us.

Important Don't ask, tell. Object-oriented programming essence is about sending order not checking state.

2.11 So far so good? Not fully!

We have a system with two classes and it offers some behavior composed out of well defined local behavior (see Figure 2-8). We can have objects composed out of other objects and messages flow within the graph. Object-oriented programming could stop here. Now it is annoying to have to duplicate structure and some methods between files and directories and this is what we will see when we will look at inheritance in Chapter **??**. Inheritance is a mechanism to specialize incrementally classes from other classes.

2.12 Conclusion

- A class describes the state (instance variables) and the behavior (methods) of all its instances. The state of an instance is the value of its instance variables and it is specific to one single object while the behavior is shared by all the instances of a class.
- Different objects, instances of different classes, can react differently to the same messages.
- When sending a message, the associated method is found and executed.

Bibliography