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

Extending superclass behavior

In the previous chapter we saw that inheritance allows the programmer to factor out and reuse state and behavior. As such inheritance supports the definition of class hierarchy where subclasses specialize behavior of their superclass. We saw that the method look up starts in the class of the receiver and goes up the inheritance chain. We explained that the method found by the lookup is then executed on the receiver of the initial message. Finally we showed that a subclass can specialize and override the behavior of its superclass by defining locally a method with the same name than one method of its superclass.

Now inheritance mechanism is even more powerful. With inheritance we can extend locally the behavior of a superclass while reusing it. It is then possible to override a method and in addition to invoke the behavior of the superclass from within the overridden method.

We will continue to use and improve the example of file and directories.

1.1 Revisiting printOn:

When we look at the following `printOn:` methods defined in the classes `MFDirectory` and `MFFile` we see that there is code repetition (as shown in Figure 1-1).

Here is the repeated code snippet.

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

Here is the definition in the two classes:

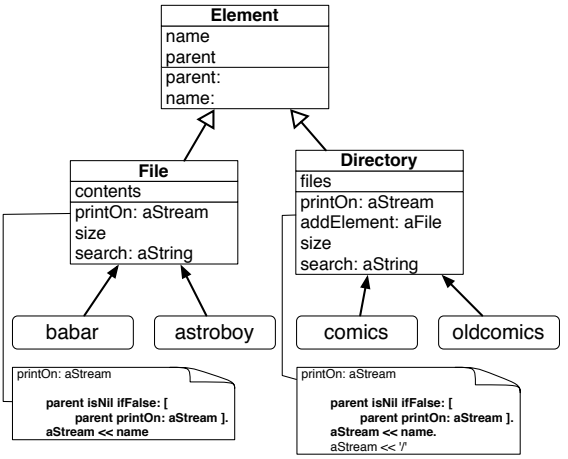

Figure 1-1 `MFFile` and `MFDirectory` contain duplicated logic in `printOn:`.

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

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

It means that if we define a new subclass we will have probably duplicate the same expression.

1.2 Improving the situation

To improve the situation above we move up the definition of the `MFFile` class because it also works for `MFElement` (as shown in Figure 1-2).

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

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

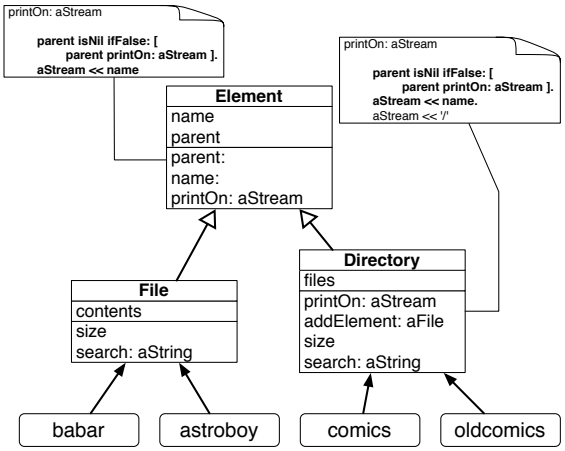

Figure 1-2 Improving the logic (but not fully).

It means that when we will add a new subclass, this class will at least have a default definition for the `printOn:` method.

Now the duplication of logic is not addressed. The same code is duplicated between the class `MFElement` and `MFDirectory`. What we see is that even if the method `printOn:` of class `MFDirectory` is overriding the method of its superclass, we would like to be able to invoke the method of the superclass `MFElement` and to add the behavior `aStream << '/'`.

Why self does not work!

The following definition does not work because it introduces an endless loop. Indeed, since the method lookup starts in the class of the receiver and `self` represents the receiver, it will always find the same method and will not be able to access the method of the superclass.

```
MFDirectory >> printOn: aStream
  self printOn: aStream.
  aStream << '/'
```

Let us make sure that you are fully with us. Imagine that we have the following expression:

```
| 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.
String streamContents: [:s | p printOn: s ]
```

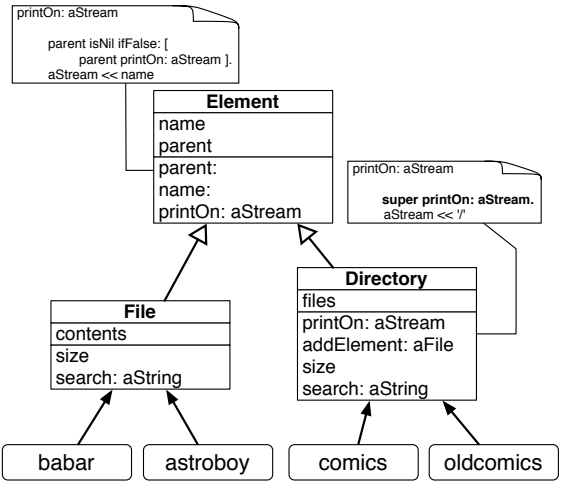

Figure 1-3 Using `super` to invoke the overridden method `printOn:`.

1. We get the message `p printOn: s`.
2. The method `printOn:` is looked up starting in the class of `p`, i.e., `MFDirectory`.
3. The method is found and applied on `p`.
4. The message `self printOn: aStream` is about to be executed.
5. The receiver is `self` and represents `p`. The method `printOn: aStream` is looked up in the class of the receiver, i.e., `MFDirectory`.
6. The same method is found in the class `MFDirectory` and the process restarts at point 3.

In summary, we would like that while doing an override, to use the behavior we are overriding. This is possible as we will see in the following section.

1.3 Extending superclass behavior using super

Let us implement the solution first and discuss it after. We redefine the method `printOn:` of the class `MFDirectory` as follows and shown in Figure 1-3.

```
MFDirectory >> printOn: aStream
  super printOn: aStream.
  aStream << '/'
```

What we see is that the method `printOn:` does not contain anymore the duplicated expressions with the method `printOn:` of the superclass (`MFElement`). Instead by using the special variable `super` the superclass method is invoked. Let us look at it in detail.

- The method `MFDirectory >> printOn:` overrides the method `MFElement`: it means that during the lookup (activated because the message `printOn:` has been sent to instances of `MFDirectory` or future subclasses), the method `MFElement >> printOn:` cannot be directly found. Indeed when a message is sent to an object, the corresponding method starts in the class of the receiver, therefore the method in `MFDirectory` is found.
- Using the special variable `super`, the method lookup is different than with `self`. When the expression `super printOn: aStream` is sent, the lookup does not start anymore from the class of the receiver, it starts *from the superclass of the class containing the expression* `super printOn:`, i.e. `MFElement`, therefore the method of the superclass is found and executed.
- Finally, `super` like `self` represents the receiver of the messages (for example an instance of the class `MFDirectory`). Therefore the method is found in the class `MFDirectory` and executed on the original object that first received the message.

Let us make sure that you are fully with us. You can compare with the previous execution simulation.

```
| 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.
String streamContents: [:s | p printOn: s ]
```

1. We get the message `p printOn: s`.
2. The method `printOn:` is looked up starting in the class of `p`, i.e., `MFDirectory`.
3. The method is found and applied on `p`.
4. The message `super printOn: aStream` is about to be executed.
5. The receiver is `super` and represents `p`. The method `printOn: aStream` is looked up in the superclass of the class containing the expression. The class containing the method is `MFDirectory`, its superclass is then `MFElement`. The lookup starts from `MFElement`.
6. The method is found in the class `MFElement` in the class.
7. The message `parent isNil` is treated on the receiver `p`.

What we see is that using `super`, the programmer can extend the superclass behavior and reuse by involving it.

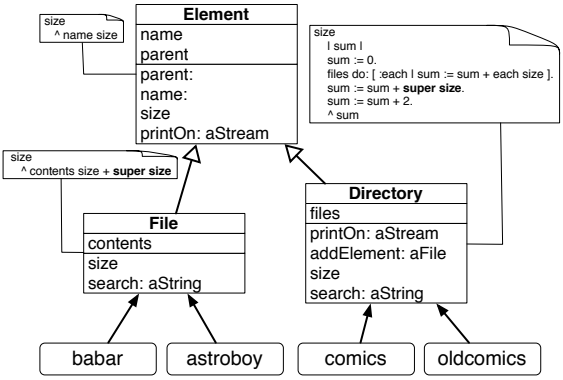

Figure 1-4 Using `super` to invoke the overridden method `size`.

> **Important** `super` is the receiver of the message but when we send a message to `super` the method lookup starts in the superclass of **the class containing** the expression `super`.

1.4 Another example

Before explaining with a more theoritical scenario *super* semantics, we want to show another example that illustrates that super expressions do not have to be the first expression of a method. We can invoke the overridden method at any place inside the overriding method.

The example could be more realistic but it shows that super expression does not have to be the first expression of a method.

Let us check the two definitions of the two methods `size` in `MFDirectory` and `MFFile`, we see that `name size` is used in both.

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

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

What we can do is the following: define `size` in the superclass and invoke it using `super` as shown in Figure 1-4. Here is then the resulting situation.

```
MFElement >> size
  ^ name size
```

```
MFFile >> size
  ^ contents size + super size
```

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

What you see is that messages sent to `super` can be used anywhere inside in the overriding method and their results can be used as any other messages.

1.5 Really understanding super

To convince you that `self` and `super` points to the same object you can use the message `==` to verify it as follows:

```
MFFile >> funky
  ^ super == self
```

```
MFFile new funky
>>> true
```

> **Important** `super` is a special variable: `super` (just like `self`) is the receiver of the message!

Now we take some time to look abstractly at what we presented so far. Imagine a situation as illustrated by Figure 1-5.

```
A new bar
>>> ...
C new bar
>>> ...
D new bar
>>> ...
```

Solution

The solutions are the following ones:

```
A new bar
>>> 10
C new bar
>>> 20
D new bar
>>> 100
```

Let us examine the evaluation of the message `aD bar`:

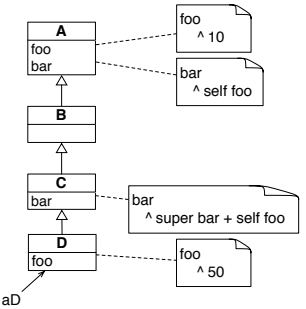

Figure 1-5 Example to understand `super`.

1. `aD`'s class is `D`.
2. There is no method `bar` in `D`.
3. The method look up in `C`. The method `bar` is found.
4. The method `bar` of `C` is executed.
5. The message `bar` is sent to `super`.
6. `super` represents `aD` but the lookup starts in the superclass of the class containing the expression `super` so it starts in `B`.
7. The method `bar` is not found in `B`, the lookup continues in `A`.
8. The method `bar` is found in `A` and it is executed on the receiver i.e., `aD`.
9. The message `foo` is sent to `aD`.
10. The method `foo` is found in `D` and executed. It returns 50.
11. Then to finish the execution of method `bar` in `C`, the rest of the expression `+ self foo` should be executed.
12. Message `self foo` returns 50 too, so the result returns 100.

Important The difference between `self` and `super` is that when we send a message to `super` the method lookup starts in the superclass of the class containing the expression `super`.

1.6 Conclusion

In this chapter we saw that inheritance also supports the possibilities to override a method and from this overriding method to invoke the overridden one. This is done using the special variable `super`. `super` is the receiver of the message like `self`. The difference is that the method lookup is changed when messages are sent to `super`. The method is looked up in the superclass of the class containing the message sent to `super`.

Bibliography