Decorator

24

The Decorator pattern – another structural pattern – adds responsibilities to objects. In the last chapter, you saw how the proxy extends objects by responsibilities. Remember – the actual object to be addressed and the proxies implemented the same interface; a proxy references another object without knowing whether it is another proxy or the desired object. You'll take a similar approach with the decorator. When I present you with an example in a moment, you may not find the difference between a proxy and a decorator clear at all – but I will discuss that below.

24.1 Build Cars

Take a look at what a car dealer has to offer. There are a few basic models with what feels like a thousand trim levels. Model A is the rolling shopping bag, Model B the down-to-earth mid-size car and Model C the upmarket variant. In addition, each model comes with an optional navigation system, air conditioning, leather upholstery, etc. Let's consider how you might code this situation.

24.1.1 One Attribute for Each Optional Extra

One could first come up with the idea of providing an attribute for each feature. If the attribute is zero, it is not desired; otherwise, it has been ordered for it.

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_24

```
public class ModelA {
    private AC ac = null;
    private GPS gps = null;

    // … abridged

    public void setAC(AC ac) {
        System.out.println("The customer buys A/C ");
        this.ac = ac;
    }

    // … abridged
}
```

The disadvantage is obvious: Each object drags around a lot of unnecessary data that has nothing to do with its actual core business. There is too much duplicate code, which makes the project error-prone and not very maintenance-friendly. When changes are due, existing code has to be drilled up. The **OpenClosed Principle** has obviously been violated.

24.1.2 Extending with Inheritance

Another approach might be to specialize, because a car with air conditioning is a car. For example, ModelA_Climate could be defined as follows:

```
public class ModelA {
    // Properties of Model A
}
public class ModelA_AC extends ModelA {
    // additional features of an air conditioner
}
```

What would be the consequence? The number of subclasses would explode because you would need to cover every conceivable combination. For example, you would need the following classes: `ModelA`, `ModelA_With_AC`, `ModelA_With_GPS`, `ModelA_With_AC_And_GPS`, `ModelB`, `ModelB_With_AC`, etc. With three basic models with three optional extras, that's 3 * 2 ^ 3 = 24 subclasses. Earlier, we talked about preferring composition to inheritance – and again, it shows that inheritance is not the means to an end. In this admittedly contrived example, the weakness of inheritance is obvious. However, I would like to assume that in practice there are cases where inheritance less evidently leads to a rigid inflexible system.

24.1.3 Decorating According to the Matryoshka Principle

Do you know the Matryoshka principle? You have a beautifully painted wooden doll that encloses another wooden doll, which in turn encloses another wooden doll, which itself encloses a wooden doll, and so on. That's kind of how the Decorator works. Each new feature encloses ("wraps") the previously assembled product. Sounds complicated? It isn't at all.

But please note that in contrast to the composite pattern from Chap. 12, we do not assemble branched structures here (this would then degenerate into a similar number of variants as in the approach with inheritance), but rather close the components "around" each other. This approach gives us the flexibility to add optional extras to the basic model in any order and frequency.

24.1.3.1 Defining the Basic Models

Basic models and optional extras implement the same interface: Component. This interface describes what each item must be able to do, namely name its price and describe itself. Open the sample project AutoCatalog. You will find the interface `Component` in the package `commons`.

```
public interface Component {
    public int getPrice();
    public String getDescription();
}
```

In the package `basicmodels` you will find different models, for example `ModelC`.

```
public class ModelC implements Component {
    @Override
    public int getPrice() {
       return 50000;
    }

       @Override
       public String getDescription() {
          return "A car of the upper middle class";
    }
}
```

In the following paragraph, you will define the optional extras.

24.1.3.2 Defining the Optional Extras

The optional extras are intended to inherit from the common upper-tier optional extra, which is itself also of type Component and holds a reference to another Component.

```
public abstract class OptionalEquipment
                                implements Component {
    protected final Component basicComponent;

    protected OptionalEquipment(Component component) {
        this.baseComponent = component;
    }
}
```

All special features define the methods of the interface Component. They access the data of the referenced component and add or concatenate their own value.

```
public class AC extends OptionalEquipment {
    public AC(Component component) {
        super(Component);
    }

    @Override
    public int getPrice() {
       return basicComponent.getPrice() + 500;
    }

    @Override
    public String getDescription() {
        return basisComponent.getDescription() +
                                  " and air conditioning";
    }
}
```

The following paragraph shows you how to put the individual components together.

24.1.3.3 The Client Plugs the Components Together

The main method of the `ApplStart` class first creates the base model and wraps it with a navigation system. Afterwards the navigation system is wrapped with an air conditioner. Since the client now has no further requests, you can call `getDescription()` and `getPrice()` on the air conditioner as the outermost wrapper.

```
public class ApplStart {
    public static void main(String[] args) {
        Component basicModel = new ModelC();
        Component gps = new GPS(basicModel);
        Component ac = new AC(gps);
        System.out.println("CustomerRequest: \n\t" +
```

```
                    ac.getDescription() + "\nPrice: \n\t"
                    + ac.getPrice());
        }
    }
```

The console outputs:

```
   Customer requirement:
       A car of the upper middle class and a satellite navigation and
air conditioning
    Price:
       51380
```

You can plug together the basic models and the equipment variants as you wish and, above all, flexibly. Even the number of components no longer plays a role. Theoretically, you could fit two air conditioners and three navigation units – if that's what the customer wants. The code is robust and maintainable because each class has a single function. High cohesion is an indication of proper design. If the price of the navigation system goes up, you only have to change the code in one place. And finally, let me introduce Decorator terminology: The interface component is a component. Each base model is a concrete component. The interface of the optional extras is a decorator; the optional extras themselves are the concrete decorators.

> The realization of the pattern is similar to the realization of the proxy pattern. With the Proxy I offered you the example of the UnmodifiableList. In the internet and in the literature, you can find this example partly also with the Decorator. Is there something wrong? Where does this class belong? To come to an answer, I have to think about the task of the UnmodifiableList. I also need to keep in mind the goal of the Patterns. The UnmodifiableList controls access to a list. When I sort it in at the Proxy, I emphasize the controlling nature; a Proxy controls access to the actual system, perhaps even restricts it. The Decorator, on the other hand, adds functionality to a system without changing it itself. Therefore, the UnmodifiableList is clearly a proxy to me.

In the next paragraph you will learn where you have already encountered the Decorator in practice.

24.2 Practical Examples

Use the Decorator Pattern to add scroll bars to Swing components and send data via streams.

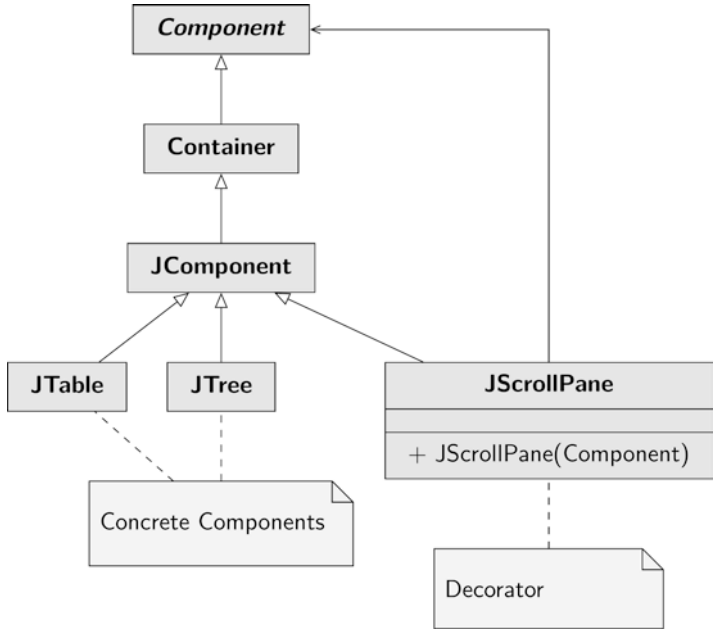

Fig. 24.1 Class hierarchy of the Swing components around JComponent (simplified)

24.2.1 The "JScrollPane" Class

You can add scroll bars to any component: a JTree, a JPanel, a JTable, and so on. In doing so, you plug the component together like the matryoshka and add it to a container.

```
JFrame frmDisplay = new JFrame();
JTable tblData = new JTable();
JScrollPane scrTable = new JScrollPane(tblData);
frmDisplay.add     (scrTable);
```

A table with scrollbars is displayed on the JFrame. You can plug the components into each other because they are all of type `JComponent` and the constructor of JScrollPane expects an object of supertype `Component`. The interaction can be found in Fig. 24.1.

24.2.2 Streams in Java

When you start looking at streams, you see the drawback of the Decorator Pattern. You have a multitude of classes that, at first glance, all do the same thing. Looking at the `java.io` package doesn't really make any programmer happy. With knowledge of the Decorator Pattern, you can bring order to the hodgepodge of classes.

24.2.2.1 Streams as Decorator

Streams connect data sources and destination. Data to be transmitted can be either bytes or characters. The inheritance hierarchies are different in each case: ByteStreams are processed by the abstract superclasses `InputStream` and `OutputStream`, CharacterStreams by `Reader` and `Writer`.

The `FilterInputStream` class is the superclass for the Decorator classes. An object of this instance references another InputStream and overrides all methods so that any request is forwarded to the referenced object. The API documentation for the FilterInputStream states:

> A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream. Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

If you read through the quote, you will find exactly the purpose description of the Decorator Pattern.

The idea is to extend the functionality of objects without creating subclasses. For example, a subclass of FilterInputStream is BufferedInputStream, which writes the data to a buffer, which improves performance. The API documentation describes the task like this:

> A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input …

The `FileInputStream` and `ByteArrayInputStream` classes are the components to be decorated; they both inherit from `InputStream`. The `FileInputStream` class connects to a file and `ByteArrayInputStream` expects an array of bytes as the data source. Both classes can read bytes from the data source and decorate it with a BufferedInputStream. You could decorate a zipped file with another decorator, the `ZipInputStream`. To do this, you plug the objects together like nesting dolls.

```
InputStream in =
   new ZipInputStream(
         new BufferedInputStream(
             new FileInputStream( /* filename */ )));
```

The inheritance hierarchy after the `OutputStream` class is set up accordingly. For example, you have a FileOutputStream there that connects to a file that you write data to. The Decorator interface is represented by the `FilterOutputStream` class. And from this class, BufferedOutputStream is derived:

> By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.

With what you have read in this paragraph you can read and write a file byte by byte. The code of such a small copy program can be found in the sample project Streams_1.

```
public static void main(String[] args) {
    var chooser = new JFileChooser();
    var returnValue = chooser.showOpenDialog(null);
    if (returnValue == JFileChooser.APPROVE_OPTION) {
        var input = chooser.getSelectedFile();
        returnValue = chooser.showSaveDialog(null);
        if (returnValue == JFileChooser.APPROVE_OPTION) {
               var output = chooser.getSelectedFile();
               try ( var in = new FileInputStream(input);
                   var out = new FileOutputStream(output)) {
                       int i;
                       while ((i = in.read()) != -1)
                           out.write(i);
               } catch (IOException ex) {
                   ex.printStackTrace();
               }
           }
       }
 }
```

So far all is well, you can verify the Decorator pattern in the API.

24.2.3 Streams as Non-Decorator

You could also go one step further and wrap the FileOutputStream in a DataOutputStream to be able to output primitive data types. In this way, you extend the functionality of the FileOutputStream – in the sense of the Decorator Pattern – without changing it. You can find a corresponding example in the sample project Streams_2.

```
private static void write() {
    final var file = new File("person.txt");

    try ( var fos = new FileOutputStream(file);
                var dos = new DataOutputStream(fos);) {
          var id = 4711;
          var salary = 2466.77;
          var isBoss = true;
          dos.writeInt(id);
          dos.writeDouble(salary);
          dos.writeBoolean(isBoss);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

And accordingly read out the file again.

```
 private static void read() {
     final var file = new File("person.txt");

     try ( var fis = new FileInputStream(file);
                var dis = new DataInputStream(fis);) {
           var id = dis.readInt();
           var salary = dis.readDouble();
           var isBoss = dis.readBoolean();
          System.out.println("The employee with the personnel number
" + id + " earns " + salary + " Euro");
           if (isChef)
                System.out.println("He's the boss");
     } catch (IOException e) {
           e.printStackTrace();
     }
  }
```

Then when a method calls write() and read(), it prints to the console:

```
The employee with the personnel number 4711 earns 2466.77 Euro
He's the boss
```

Why is this construct not a decorator? The decorator pattern allows another object to extend the interface. However, the client code must not be affected by this; decoration must be transparent to the client. In this example, however, that would be exactly the case – the specialized methods cannot be called on an object of type `InputStream`, or `OutputStream`. If the programmer wants to make use of this, he must program against the interface of the `DataInput/OutputStream`.

Streams are mentioned in the Decorator Pattern context in almost every Patterns book – including the GoF. That's true as long as you don't need to change the interface. I have repeatedly read (quoting mutatis mutandis): "Everything that is IO is also Decorator." I don't think such an absolutely worded statement is tenable.

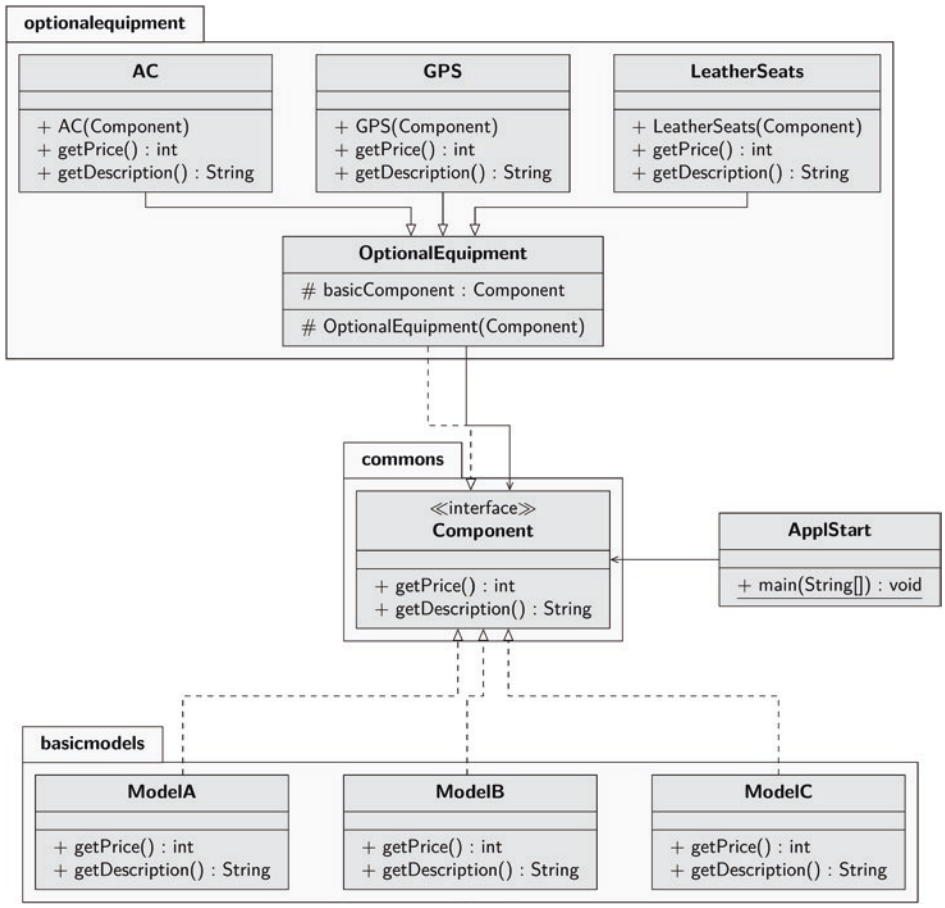

Fig. 24.2 UML diagram of the Decorator Pattern (example project AutoCatalog)

24.3 Decorator – The UML Diagram

This time you can see the UML diagram in Fig. 24.2 from the first example project AutoCatalog.

24.4 Summary

Go through the chapter again in key words:

- You need to add responsibilities to objects.
- Inheritance leads you into a rigid, inflexible, and unmaintainable system.

- The extension should not change the object to be extended.
- The Decorator Pattern provides an alternative to subclassing.
- In the Decorator, there are components that are decorated.
- The Decorator classes implement the same interface as the components.
- The decorator references the object to be decorated.
- Decorators can have their own methods, but must adhere to the component interface.
- The advantage is a stable, flexible and cohesive system.
- The disadvantage is that many similar classes are formed.
- The difference with the proxy is that the proxy controls the object, but the decorator adds functionality to the object.
- Unlike the composite, the decorator does not specify a fixed dependency and frequency.
- In contrast to the adapter, the interface of the Decorator is identical in both directions – so that further Decorators can also be connected.

24.5 Description of Purpose

The Gang of Four describes the purpose of the "Decorator" pattern as follows:

> Dynamically extend an object with responsibilities. Decorators provide a flexible alternative to subclassing to extend the functionality of a class.