DESIGN PATTERNS

The Decorator Pattern in Depth

Add functionality to a class without modifying it.

by Ian Darwin

Interior decorators are people who come to your house and tell you how to modify the look and feel of your rooms—for example, what furniture and wall or floor covering you should add. Decorators in software are pieces of software that add functionality to code by "wrapping" a target class without modifying it. They exemplify the open/closed principle, which states that classes should be open for extension but closed to modification.

Consider the set of classes needed to manage orders for a small art photography business. The business sells prints, which can be made in various sizes, on matte or glossy paper, with or without a frame, without a mat or with one or more mats that come in many colors. It also sells prints digitally through a variety of stock photo agencies.

Your first thought might be to use inheritance to implement this data model. However, trying to make a different class for every combination of paper finish, paper size, with and without a mat, and with and without a frame would result in a true explosion of classes. And it would all fall apart as soon as market conditions changed and the business tried to add another variable. The opposite approach—trying to make one class to handle all the combinations—would result in a tangled mess of `if` statements that are fragile and difficult to understand and maintain.

One of the best solutions to this problem is the *Decorator* pattern, which allows you to add functionality to an existing component *from outside* the class. To do this, you create a class called `Decorator` (typically abstract), which needs to implement the same interface as the original `Component` (the class to which we're adding functionality); that is, the `Decorator` needs to be able to substitute for the original, using its interface or class definition as appropriate. The `Decorator` will hold an instance of the `Component` class, which it is also extending. The `Decorator` will, thus, be in both a *has-a* and an *is-a* relationship with the `Component`. So you'll often see code like this in the `Decorator`:

```
public abstract class Decorator extends Component {
        protected Component target;
                . . .
        }
```

As a result of this inheritance, the `Decorator` or its subclasses can be used where the original was used, and it will directly delegate business methods to the original. You can then make subclasses from this `Decorator`. The subclasses will usually add "before" and/or "after" functionality to some or all of the methods that they delegate to the real target; this is, in fact, their raison d'être.

In the photo business example, the original `Component` is `PhotoImage`. Its constructor takes the artistic name of the image and the name of the

file in which the master version of the sellable image is stored. The code that follows can be found online.

The `Decorator` class both extends `PhotoImage` and has a photo. You can then have `Print` and `DigitalImage` decorators for the two main ways of selling. The `Print` constructor represents a physical print, so it has a paper size (width and height, in inches). For `Print`, you can have `Frame`, `Mat`, and other `Decorator`s, with parameters as needed (for example, mats have color).

First, here are a few examples of using this set of classes, from `ImageSales.java`:

```
// Create an undecorated image
    final PhotoImage image = new PhotoImage(
        "Sunset at Tres Ríos", "2020/ifd12345.jpg")

    // Make a print of that, on US letter-size pape
    Print im1 = new Print(11, 8.5, image);
    addToPrintOrder(im1);

    // Make a 19x11 print of a second image, matted
    Print im2 =
        new Print(19, 11,
            new Frame(
                new Mat("Lime Green",
                    new PhotoImage("Goodbye at the
                        "1968/ifd.00042.jpg"))));
    addToPrintOrder(im2);

    // Make a digital print sale
    PhotoImage dig = new DigitalImage(image, StockA
System.out.println(dig);
```

The `addToPrintOrder()` method takes a `Print` argument, providing a degree of type safety. In this simple demo, this method just prints the argument by calling `toString()`, to show the effect of the delegation methods.

Here is the code for `PhotoImage` (the `Component`):

```
/** A PhotoImage is a picture that I took at some p
     */
    public class PhotoImage {
        /** The human-readable title */
        String title;

        /** Where the actual pixels are */
        String fileName;

        /** How many pixels are there in the image
        int pixWidth, pixHeight;

        public PhotoImage() {
            // Empty; used in Decorators
        }

        public PhotoImage(String title, String file
            super();
            this.title = title;
            this.fileName = fileName;
        }

        /** Get a printable description; may be mor
         * but in any case, it's the example delega
         */
        public String getDescription() {
            return getTitle();
        }

        /** Default toString() just uses getDescrip
        @Override
        public String toString() {
            return getDescription();
        }
```

```
        // setters and getters...
}
```

Here is part of the code for `ImageDecorator`, which is the `Decorator` class:

```
public abstract class ImageDecorator extends PhotoIm
        protected PhotoImage target;

        public ImageDecorator(PhotoImage target) {
            this.target = target;
        }

        @Override
        public String getDescription() {
            return target.getDescription();
        }

        @Override
        public String toString() {
            return getDescription();
        }
}
```

Here is part of one of the decorators, the `Print` decorator:

```
/** A Print represents a PhotoImage with the physica
 */
public class Print extends ImageDecorator {
    /** PrintWidth, PrintHeight are in inches for th
    private double printWidth, printHeight;

    public Print(double printWidth, double printHeig
        super(target);
        this.printWidth = printWidth;
        this.printHeight = printHeight;
    }

    @Override
    public String getDescription() {
        return target.getDescription() + " " +
            String.format("(%4.1f x %4.1f in)",
                          getPrintWidth(), getPrintH
    }
        // setters and getters...
}
```

One of the key parts of this code is how the `Print` class's `getDescription()` calls the target's method of the same name and also adds its own functionality to it. This is the "delegate but also add functionality" part of the pattern. And it's why, when you run the main program, you get this output for the `Print` object:

```
Goodbye at the Station, Matted(Lime Green), Framed
```

It might seem unusual to put the secondary characteristics (paper size, mat color) as the first arguments to the constructors. If you're as compulsive as I am about such things, you'd normally put the principal element—the `Component`—as the first argument. However, in practice, putting the secondary items first facilitates the coding style with which multiple decorators are typically combined. Take a look back at the main program and imagine lining up the brackets and closing arguments when you have that many nested constructor calls. But if you don't like it, that's fine; it's just a style issue. Picking one way and being consistent in your hierarchy will make your life easier.

If the Decorator pattern looks familiar to you, it should. This is the same way that the common java.io Streams and Readers and Writers have worked since the beginning of Java.

The generic view of the class hierarchy is shown in **Figure 1**, where `Component` is the `PhotoImage`, `Print` and `DigitalImage` subclass `Decorator`, and `Mat` and `Frame` are other decorators. The only operation illustrated is `getDescription()`; others would exist in the production code.

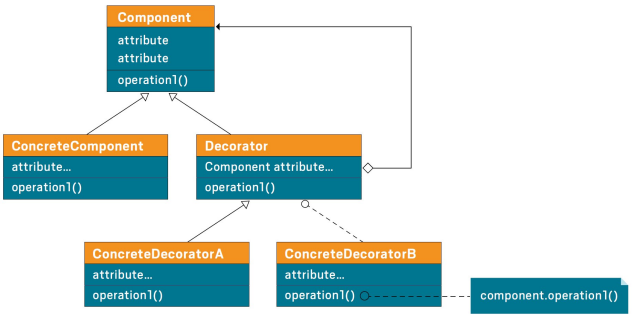

Figure 1. Decorator hierarchy

There is no special reason that `Print` and `DigitalImage` must be decorators rather than the `ConcreteComponent` shown in **Figure 1**—that is, subclassing `PhotoImage` directly. The reason I made them decorators is so that I can, as in the code sample, create one `PhotoImage` and use it to create both a `Print` and a `DigitalImage`. You might not need to do the equivalent operation in your class hierarchy; it depends on your use cases as to whether you make classes like that be `ConcreteComponent`s or decorators. Note also that for very simple cases, such as where you will only ever have one decorator class, you could skip the definition of the `Decorator` as a separate abstract class and do the delegation directly in your actual decorator. The separate class is just a convenient place to build a hierarchy when you expect you will have multiple decorators.

All this code is on GitHub, in the subdirectory `src/main/java/structure/decorator`.

Using Frameworks

Various widely used frameworks provide mechanisms for intercepting calls on managed beans. For example, Java EE's interceptors and Spring's aspect-oriented programming (AOP) offer both predefined interceptors and the ability to define your own. These are decorators. For example, both frameworks provide annotation-based transaction management, in which you can say something like this:

```
public class WidgetService {

        @Transactional
        public void saveWidget(Widget w) {
            entityManager.persist(w);
        }
}
```

Behind the scenes, both frameworks provide an interceptor whose pseudocode is probably akin to the following. The real thing will be more complex because of different transaction types (read-only, read-write) and different needs (transaction required, new transaction required, etcetera), different handling of checked versus unchecked exceptions, and so on.

```
public class TransactionInterceptor {

        UserTransaction transaction;    // Injected

        // PSEUDOCODE

        public void interceptTransaction(
```

```
            Method method, Object target, Object[] a

            if (transaction.getStatus() != Status.N
                transaction.begin();
            }
            try {
                method.invoke(target, args);
                transaction.commit();
            } catch (Exception ex) {
                transaction.rollback();
                throw ex;
            }
        }
}
```

You can write your own interceptors using the EJB3 interceptor API (see `@Interceptor`) or Spring's AOP (see `@Around`).

Border Decorator

Historically, many of the earliest uses of the Decorator pattern were graphical add-ons, for example, to add a nice border to a text area or dialog box as part of a layout. Decorators were used this way in many early and influential window systems, including Smalltalk and InterViews. You'd use these as something like the following pseudocode (the `TextField` constructor takes row and column arguments):

```
Component niceTextBox = new BorderDecorator(new Text
    Component niceTextArea =
        new BorderDecorator(new ScrollingDecorator(
```

Depending on your application, you might or might not need to forward all delegated methods. An issue with `Decorator` is that if the component being decorated has a large number of methods (it shouldn't, if you remembered to keep things simple), and you need to forward all the methods, the number of delegation methods required can make this pattern cumbersome. Yet if you don't forward a particular inherited method, because of how inheritance works that method will (if called) execute in the context of the `Decorator` alone, not the target, and this can lead to bad results. You can use the IDE to generate all the delegates (for example, in Eclipse, using the `Source → Generate Delegate Methods` option), but that does not lead to maintainable code. The Strategy pattern or the Proxy pattern might be a good alternative way of adding functionality.

The people who wrote Java's Swing UI package were aware of this issue. They wanted to decorate `JComponent` (a subclass of the Abstract Window Toolkit's `Component`), but that dear thing has around 120 public methods. To allow decoration in the original sense, without making you subclass it just for this purpose, they took a variant approach and provided the Swing `Border` object. This `Border` object isn't used like a traditional decorator but as what you might roughly call a "plugin decorator," using these methods defined in `JComponent`:

```
public void setBorder(Border);
public Border getBorder();
```

This `Border` class is in the `javax.swing.border` package. You get instances of `Border` from `javax.swing.BorderFactory` by calling methods such as `createEtchedBorder()` and `createTitledBorder()`.

I/O Streams

If the Decorator pattern looks familiar to you, it should. This is the same way that the common `java.io Streams` and `Reader`s and `Writer`s

have worked since the beginning of Java. You've probably seen code like this a million times:

```
// From IOStreamsDemo.java
    BufferedReader is =
        new BufferedReader(new FileReader("some file
    PrintWriter pout =
        new PrintWriter(new FileWriter("output file
    LineNumberReader lrdr =
        new LineNumberReader(new FileReader(foo.getN
```

Here, there is no separate `Decorator` class; the classes involved are all just subclass `Reader` or `Writer` (or `InputStream` and `OutputStream` for binary files) directly, and all the classes have constructors that accept an instance of the top-level class (or any subclass, of course) as an argument. But this usage fits in with the basic description of the Decorator pattern: one class adds functionality to another by wrapping it.

Decorator Versus Proxy

Proxy is another pattern in which classes expand upon other classes, often using the same interface as the object being proxied. As a result, people sometimes confuse the Proxy pattern with the Decorator pattern. However, Decorator is primarily about adding *functionality* to the target. The Gang of Four definition of Proxy is that it's about *controlling access* to the target. This control could be to provide lazy creation for expensive objects, to enforce permissions or a security-checking point of view (a security proxy), or to hide the target's location (such as a remote access proxy as used in RPC-based networking APIs such as RMI, remote EJB invocation, the JAX-RS client, or the JAX-WS client).

In the lazy-creation situation, a lightweight proxy is created with the same interface as the expensive (heavyweight) object, but none or only a few of the fields are filled in (ID and title, perhaps). This proxy handles creation of the heavyweight object only when and if a method that depends on the full object is called.

In the networking situation, the client code appears to be calling a local object, but it is in fact calling a proxy object that looks after the networking and the translation of objects to and from a transmissible format, all more or less transparently. With the Decorator pattern, the client is usually responsible for creating the decorator, often at the same time that it creates the object being decorated.

Conclusion

Decorators are a convenient way of adding functionality to a target class without having to modify it. Decorators usually provide the same API as the target, and they do additional work before or after forwarding the arguments to the target object. Try using the Decorator pattern the next time you need to add functionality to a small set of classes.

This article was originally published in the November/December 2018 issue of *Java Magazine*.

Ian Darwin

Ian Darwin (@Ian_Darwin) is a Java Champion who has done all kinds of development, from mainframe applications and desktop publishing applications for UNIX and Windows, to a desktop database application in Java, to healthcare apps in Java for Android. He's the author of *Java Cookbook* and *Android Cookbook* (both from O'Reilly). He has also written a few courses and taught many at Learning Tree International.