THE DECORATOR PATTERN

The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class. Suppose we have a program that uses eight objects, but three of them need an additional feature. You could create a derived class for each of these objects, and in many cases this would be a perfectly acceptable solution. However, if each of these three objects require *different* modifications, this would mean creating three derived classes. Further, if one of the classes has features of *both* of the other classes, you begin to create a complexity that is both confusing and unnecessary.

For example, suppose we wanted to draw a special border around some of the buttons in a toolbar. If we created a new derived button class, this means that all of the buttons in this new class would always have this same new border, when this might not be our intent.

Instead, we create a Decorator class that *decorates* the buttons. Then we derive any number of specific Decorators from the main Decorator class, each of which performs a specific kind of decoration. In order to decorate a button, the Decorator has to be an object derived from the visual environment, so it can receive paint method calls and forward calls to other useful graphic methods to the object that it is decorating. This is another case where object containment is favored over object inheritance. The decorator is a graphical object, but it contains the object it is decorating. It may intercept some graphical method calls, perform some additional computation and may pass them on to the underlying object it is decorating.

Decorating a CoolButton

Recent Windows applications such as Internet Explorer and Netscape Navigator have a row of flat, unbordered buttons that highlight themselves with outline borders when you move your mouse over them. Some Windows programmers call this toolbar a CoolBar and the buttons CoolButtons. There is no analogous button behavior in the JFC, but we can obtain that behavior by *decorating* a JButton. In this case, we decorate it by drawing plain gray lines over the button borders, erasing them.

Let's consider how to create this Decorator. *Design Patterns* suggests that Decorators should be derived from some general Visual Component class

and then every message for the actual button should be forwarded from the decorator. In Java, this is completely impractical, because there are literally hundreds of method calls in the base JComponent class that we would have to reimplement. Instead, while we will derive our Decorator from the JComponent class, we will use its container properties to forward all method calls to the button it will contain.

Design Patterns suggests that classes such as Decorator should be abstract classes and that you should derive all of your actual working (or concrete) decorators from the abstract class. In this Java implementation, this is scarcely necessary since the base Decorator class has no public methods at all other than the constructor, since all of them are methods of JComponent itself.

```
public class Decorator extends Jcomponent {
   public Decorator(JComponent c)    {
      setLayout(new BorderLayout());
       //add component to container
      add("Center", c);
   }
}
```

Now, let's look at how we could implement a CoolButton. All we really need to do is to draw the button as usual from the base class, and then draw gray lines around the border to remove the button highlighting.

```
//this class decorates a CoolButton so that
//the borders are invisible when the mouse
//is not over the button
public class CoolDecorator extends Decorator
{
   boolean mouse_over;    //true when mouse over button
   JComponent thisComp;

   public CoolDecorator(JComponent c)
   {
      super(c);
      mouse_over = false;
      thisComp = this;      //save this component
      //catch mouse movements in inner class
      c.addMouseListener(new MouseAdapter()
      {
      public void mouseEntered(MouseEvent e) {
         mouse_over=true;     //set flag when mouse over
         thisComp.repaint();
         }
      public void mouseExited(MouseEvent e) {
         mouse_over=false;    //clear if mouse not over
         thisComp.repaint();
```

```
         }
      });

   }
   //paint the button
   public void paint(Graphics g)
   {
      super.paint(g);      //first draw the parent button
      if(! mouse_over) {
         //if the mouse is not over the button
         //erase the borders
         Dimension size = super.getSize();
         g.setColor(Color.lightGray);
         g.drawRect(0, 0, size.width-1, size.height-1);
         g.drawLine(size.width-2, 0, size.width-2,
                           size.height-1);
         g.drawLine(0, size.height-2, size.width-2,
                           size.height-2);
      }
   }
}
```

Using a Decorator

Now that we've written a CoolDecorator class, how do we use it? We simply create an instance of the CoolDecorator and pass it the button it is to decorate. We can do all of this right in the constructor. Let's consider a simple program with two CoolButtons and one ordinary JButton. We create the layout as follows:

```
super ("Deco Button");
JPanel jp = new JPanel();

getContentPane().add(jp);
jp.add( new CoolDecorator(new JButton("Cbutton")));
jp.add( new CoolDecorator(new JButton("Dbutton")));
jp.add(Quit = new JButton("Quit"));
Quit.addActionListener(this);
```

This program is shown below, with the mouse hovering over one of the buttons.

Now that we see how a single decorator works, what about multiple decorators? It could be that we'd like to decorate our CoolButtons with another decoration, say, a red diagonal line. Since the argument to any Decorator is just a JComponent, we could create a new decorator with a decorator as its argument.

Let's consider the SlashDecorator, which draws that diagonal red line:

```
public class SlashDecorator extends Decorator
{
   int x1, y1, w1, h1;     //saved size and posn

   public SlashDecorator(JComponent c)    {
      super(c);
   }
//-----------------------------------------------
   public void setBounds(int x, int y, int w, int h)    {
      x1 = x; y1= y;             //save coordinates
      w1 = w; h1 = h;
      super.setBounds(x, y, w, h);
   }
//-----------------------------------------------
   public void paint(Graphics g)    {
      super.paint(g);            //draw button
      g.setColor(Color.red);     //set color
      g.drawLine(0, 0, w1, h1); //draw red line
   }
}
```

Here we save the size and position of the button when it is created, and then use those saved values to draw the diagonal line.

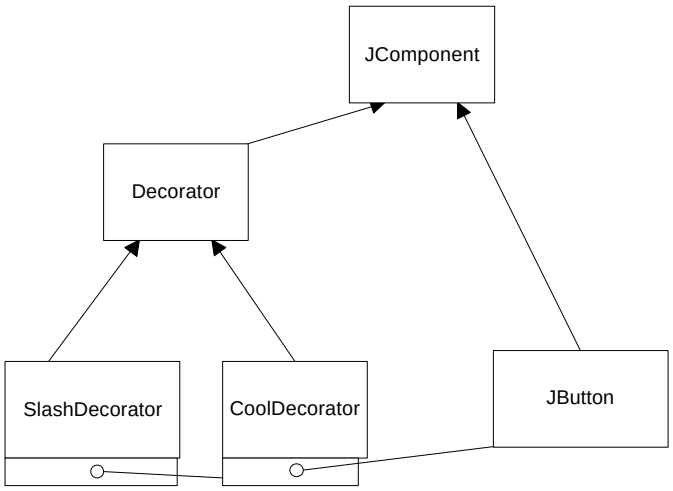
You can create the JButton with these two decorators by just calling one and then the other:

```
jp.add(new SlashDecorator(
       new CoolDecorator(new JButton("Dbutton"))));
```

This gives us a final program that displays the two buttons like this:

Inheritance Order

Some people find the order of inheritance in Decorators confusing, because we are surrounding a button with a decorator that inherits from a JComponent. We illustrate this inheritance tree below.



A JButton is a child of JComponent, and is encapsulated in a Decorator, which not only is a child of JComponent but encapsulates one as well. The JComponent it encapsulates is, in this case, a JButton.

Decorating Borders in Java

One problem with this particular implementation of Decorators is that it is not easy to expand the size of the component you are decorating, because you add the component to a container and allow it to fill the container completely. If you attempt to draw lines outside the area of this component, they are clipped by the graphics procedure and not drawn at all.

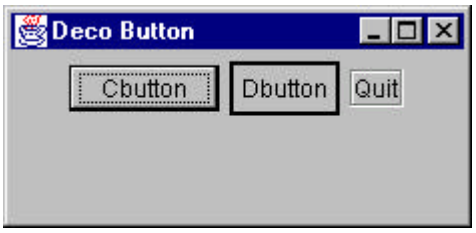
The JFC provides its own series of Border objects that are a kind of decorators. Like a Decorator pattern, you can add a new Border object to any JComponent, and there also is a way to add several borders. However, unlike the Decorator pattern, it is not a JComponent and you do not have the flexibility to intercept and change specific events.

The JFC defines several standard border classes:

BevelBorder(*n*)	Simple 2-line bevel, can be LOWERED or RAISED

CompoundBorder (*inner, outer)*	Allows you to add 2 borders
EmptyBorder(*top, left, bottom, right)*	Blank border width specified on each side.
EtchedBorder	Creates etched norder.
LineBorder(*width, color)*	Creates simple line border,
MatteBorder	Creates a matte border of a solid color or a tiled icon.
SofBeveledBorder	Creates beveled border with rounded corners.
TitledBorder	Creates a border containing a title. Use this to surround and label a JPanel.

These borders are simple to use, in conjunction with the *setBorder* method of each JComponent. The illustration below shows a normal JButton with a 2-pixel solid line border, combined with a 4-pixel EmptyBorder and an EtchedBorder.



This was created with the following simple code:

```
getContentPane().add(jp);
jp.add( Cbutton = new JButton("Cbutton"));
jp.add( Dbutton = new JButton("Dbutton"));
EmptyBorder ep = new EmptyBorder(4,4,4,4);
LineBorder lb = new LineBorder(Color.black, 2);
Dbutton.setBorder(new CompoundBorder(lb, ep));
jp.add(Quit = new JButton("Quit"));
EtchedBorder eb = new EtchedBorder();
Quit.addActionListener(this);
Quit.setBorder(eb);
```

One drawback of these Border objects is that they replace the default Insets values that determine the spacing around the component. Note that we had to add a 4-pixel EmptyBorder to the Dbutton to make it similar in size to the CButton. We did not do this for the Quit button, and it is therefore substantially smaller than the others.

Non-Visual Decorators

Decorators, of course, are not limited to objects that enhance visual classes. You can add or modify the methods of any object in a similar fashion. In fact, non-visual objects are usually easier to decorate, because there are usually fewer methods to intercept and forward.

While coming up with a simple example is difficult, a series of Decorators do occur naturally in the java.io classes. Note the following in the Java documentation:

> *The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the underlying input stream. Subclasses of FilterInputStream may further override some of these methods as well as provide additional methods and fields.*

The FilterInputStream class is thus a Decorator that can be wrapped around any input stream class. It is essentially an abstract class that doesn't do any processing, but provides a layer where the relevant methods have been duplicated. It normally forwards these method calls to the enclosed parent stream class.

The interesting classes derived from FilterInputStream include

BufferedInputStream	Adds buffering to stream so that every call does not cause I/O to occur.
CheckedInputStream	Maintains a checksum of bytes as they are read
DataInputStream	Reads primitive types (Long, Boolean, Float, etc.) from the input stream.
DigestInputStream	Computes a MessageDigest of any input stream.
InflaterInputStream	Implements methods for uncompressing data.
PushbackInputStream	Provides a buffer where data can be "unread," if during parsing you discover you need to back up.

These decorators can be nested, so that a pushback, buffered input stream is quite possible.

Decorators, Adapters and Composites

There is an essential similarity among these classes that you may have recognized. Adapters also seem to "decorate" an existing class. However, their function is to change the interface of one or more classes to one that is more convenient for a particular program. Decorators add methods to particular instances of classes, rather than to all of them. You could also imagine that a composite consisting of a single item is essentially a decorator. Once again, however, the intent is different

Consequences of the Decorator Pattern

The Decorator pattern provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class. It also allows you to customize a class without creating subclasses high in the inheritance hierarchy. *Design Patterns* points out two disadvantages of the Decorator pattern One is that a Decorator and its enclosed component are not identical. Thus tests for object type will fail. The second is that Decorators can lead to a system with "lots of little objects" that all look alike to the programmer trying to maintain the code. This can be a maintenance headache.

Decorator and Façade evoke similar images in building architecture, but in design pattern terminology, the Façade is a way of hiding a complex system inside a simpler interface, while Decorator adds function by wrapping a class. We'll take up the Façade next.