

CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Decorator Pattern Apr. 3, 2007

1

What is it?

- 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.
- The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class.

2

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.

3

Example Decorator

- 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 a 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.

4

Decorating a CoolButton

- 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 and erasing them.

5

Strategy

- 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.

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

6

Strategy

- To 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.

7

```
//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();
            }
        }
    }
}
```

8

```

    });
}
//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);
    }
}
}
}

```

9

Using the Decorator

```

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);

```



10

Another Decorator

- Next, 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.

11

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

```

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

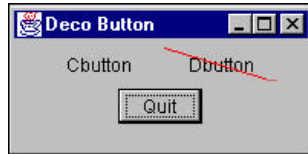
    public SlashDecorator(JComponent c)    {
        super(c);
    }
    //-----
    public void setBounds(int x, int y, int w, int h)    {
        x1 = x; y1= y;    //save coordinates
        w1 = w; h1 = h;
        super.setBounds(x, y, w, h);
    }
    //-----
    public void paint(Graphics g)    {
 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

```

12

Using the Two Decorators

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



13

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.

14

Summary

- 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 allows you to customize a class without creating subclasses high in the inheritance hierarchy.
- 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.

15