

CHAPTER 2

Structural Patterns: Decorator, Proxy, and Bridge

We start our tour of design patterns with the group known as the structural patterns. There are seven patterns that make up the structural group, each with the role of building flexibility, longevity, and security into computer software. The names of the patterns are important, so I'll introduce them immediately. They are:

- Decorator
- Proxy
- Bridge
- Composite
- Flyweight
- Adapter
- Façade

Structural patterns are concerned with how classes and objects are composed to form larger structures. Of the many purposes of the seven structural patterns, here are 10:

- Add new functionality dynamically to existing objects, or remove it (Decorator).
- Control access to an object (Proxy).
- Create expensive objects on demand (Proxy).
- Enable development of the interface and implementation of a component to proceed independently (Bridge).
- Match otherwise incompatible interfaces (Adapter).
- Reduce the cost of working with large numbers of very small objects (Flyweight).
- Reorganize a system with many subsystems into identifiable layers with single entry points (Façade).
- Select or switch implementations at runtime (Bridge).
- Simplify the interface to a complex subsystem (Façade).
- Treat single objects and composite objects in the same way (Composite).

Structural patterns can be employed while a system is being designed, or later on during maintenance and extension. In fact, some of them are specifically useful in the post-production stages of the lifecycle of a software system, when changes are introduced that were not foreseen and when even the interfaces between components need updating. Thus, sometimes when you want to add functionality, you will be working with existing classes that cannot be changed. The Decorator pattern is useful here. Alternatively, you might be able to design a whole system from scratch so that it works in a particularly usable way, with the Composite pattern.

Our exploration of the structural design patterns will span three chapters. In this chapter, we'll look at the Decorator, Proxy, and Bridge patterns. In Chapter 3, we'll tackle the Composite and Flyweight patterns, and in Chapter 4, we'll examine the Adapter and Façade patterns.

Our first three patterns provide ways of adding state and behavior dynamically, controlling the creation and access of objects, and keeping specifications and implementations separate. They do not draw upon any advanced features of C#, relying only on interfaces for structuring; however, at the end of this chapter, I introduce *extension methods* as an interesting way to implement the Bridge pattern.

For each pattern, we will examine its role, an illustration of where it might be used in programming, and its design (what the major players are and what roles they play in the pattern). We will then look at some simple code illustrating the implementation of the pattern and at a more concrete example. Finally, we will consider some additional real-world uses for the pattern and some exercises that you can work through to enhance your understanding of it. Also, at the end of this and every chapter, you will find a more detailed discussion of when the individual patterns might be used and a comparison of their intent and applicability.

Decorator Pattern

Role

The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically. The object does not know it is being “decorated,” which makes this a useful pattern for evolving systems. A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.

Illustration

As its name suggests, the Decorator pattern takes an existing object and adds to it. As an example, consider a photo that is displayed on a screen. There are many ways to add to the photo, such as putting a border around it or specifying tags related to the content. Such additions can be displayed on top of the photo, as shown in Figure 2-1.

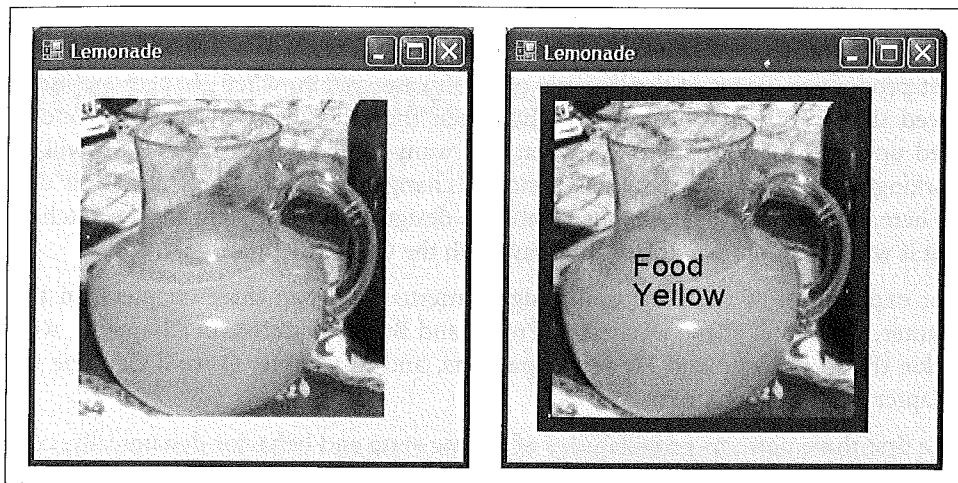


Figure 2-1. Decorator pattern illustration—(a) plain photograph and (b) photograph with tags

The combination of the original photo and some new content forms a new object. In the second image shown in Figure 2-1, there are four objects: the original photo as shown to the left, the object that provides a border, and two tag objects with different data associated with them. Each of them is a Decorator object. Given that the number of ways of decorating photos is endless, we can have many such new objects. The beauty of this pattern is that:

- The original object is unaware of any decorations.
- There is no one big feature-laden class with all the options in it.
- The decorations are independent of each other.
- The decorations can be composed together in a mix-and-match fashion.

Design

Now, we can specify the players in the Decorator pattern in a UML diagram, shown in Figure 2-2. Because this is the first pattern we are describing in UML, we'll take it slowly. (The UML that we need for patterns in this book is covered in Chapter 1 and summarized in Table 1-1.) The essential players in this UML diagram are:

Component

An original class of objects that can have operations added or modified (there may be more than one such class)

Operation

An operation in IComponent objects that can be replaced (there may be several operations)

IComponent

The interface that identifies the classes of objects that can be decorated (Component is one of these)

Decorator

A class that conforms to the IComponent interface and adds state and/or behavior (there may be more than one such class)

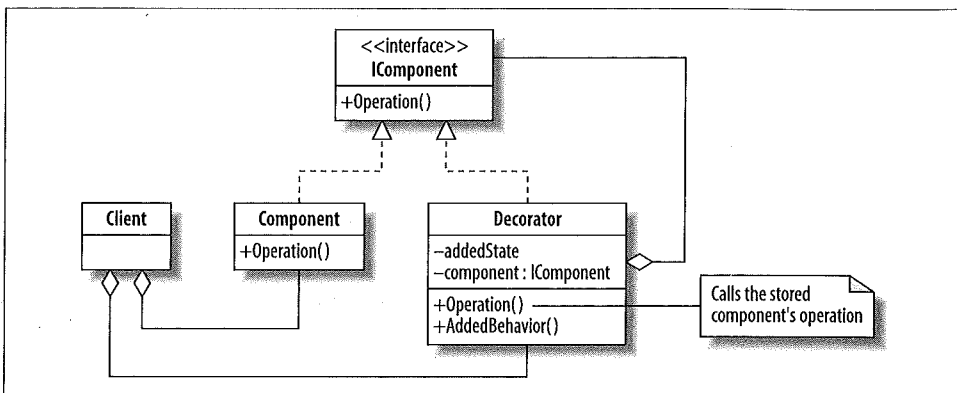


Figure 2-2. Decorator pattern UML diagram

The center of the UML diagram is the Decorator class. It includes two types of relationships with the IComponent interface:

Is-a

The is-a relationship is shown by a dotted arrow from the Decorator to IComponent, indicating that Decorator realizes the IComponent interface. The fact that Decorator inherits from IComponent means that Decorator objects can be used wherever IComponent objects are expected. The Component class is also in an is-a relationship with IComponent, and therefore the client can use Component and Decorator objects interchangeably—the heart of the Decorator pattern.

Has-a

The has-a relationship is shown by an open diamond on the Decorator, linked to IComponent. This indicates that the Decorator instantiates one or more IComponent objects and that decorated objects can outlive the originals. The Decorator uses the component attribute (of type IComponent) to invoke any replacement Operation it might wish to override. This is the way the Decorator pattern achieves its objective.

The addedBehavior operation and the addedState attribute in the Decorator class are other optional ways of extending what is in the original Component objects. We'll look at some examples momentarily.

Quiz

Match the Decorator Pattern Players with the Photo Decorator Illustration

To test whether you understand the Decorator pattern, cover the lefthand column of the table below and see if you can identify the players among the items from the illustrative example (Figure 2-1), as shown in the righthand column. Then check your answers against the lefthand column.

IComponent	Any photo
Component	A plain photo
Operation	To display a photo
Decorator	A tagged photo
Client	Creator of photos and tagged photos
IComponent	Any photo

From this list, we can see that the following would be valid statements in a Client wanting to put two tags on a photo:

```
Photo photo = new Photo();  
Tag foodTag = new Tag (photo, "Food",1);  
Tag colorTag = new Tag (foodTag, "Yellow",2);
```

By the is-a relationship, photo, foodTag, and colorTag are all IComponent objects. Each of the tags (the decorators) is created with a Component, which might be a photo or an already tagged photo. The resulting object diagram is shown in Figure 2-3. As you can see, there are actually three separate photo objects in the system. How they interact is discussed in the upcoming "Implementation" section.

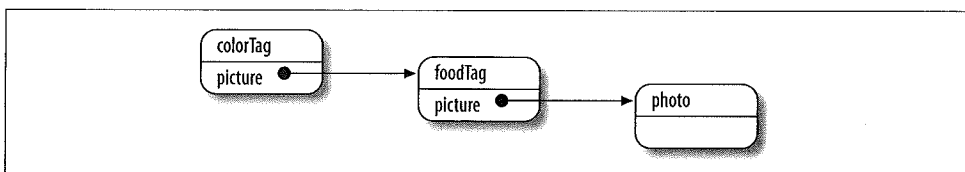


Figure 2-3. Decorator pattern objects

In most of the patterns we will encounter, the players can appear in multiple guises. To keep the UML diagrams clear and simple, not all of the options will be shown. However, we should consider the implications of these multiple players on the design of the pattern:

Multiple components

Different components that conform to the interface can also be decorated. For example, we could have a class that draws people, houses, ships, and so on from simple shapes and lines. They too could be tagged. It is for this reason that the `IComponent` interface is important, even if it does not contain any operations. In the case where we are sure there will only ever be one class of components, we can dispense with the `IComponent` interface and have the decorators directly inherit from `Component`.

Multiple decorators

We have seen that we can create different instances of a `Tag` decorator. We can also consider having other types of decorators, such as `Border` decorators or even decorators that make the photo invisible. No matter what the decorators are, each contains a component object, which might itself be a decorator, setting off a chain of changes (as suggested in Figure 2-3). Some Decorator pattern designs include an `IDecorator` interface, but it generally serves no purpose in C# implementations.

Multiple operations

Our illustration focuses on drawing as the chief operation for photos and decorations. Other examples will lend themselves to many more optional operations. Some of these will be part of the original component and its interface, whereas some will be added behaviors in certain decorators only. The client can call any of the operations individually on any of the components (decorated or otherwise) to which it has access.

Implementation

The Decorator pattern's key feature is that it does not rely on inheritance for extending behavior. If the `Tag` class had to inherit from the `Photo` class to add one or two methods, `Tags` would carry everything concerned with `Photos` around with them, making them very heavyweight objects. Instead, having the `Tag` class implement a `Photo` interface and then add behavior keeps the `Tag` objects lean. They can:

- Implement any methods in the interface, changing the initial behavior of the component
- Add any new state and behavior
- Access any public members via the object passed at construction

Before we carry on with the photo example, consider the theoretical version of the Decorator pattern in Example 2-1. Short examples such as this one are useful for showing the interaction between classes and objects of a pattern in a direct mapping to the UML. Once we move on to a real-world example, optimizations and extensions might be employed that make it more difficult to detect and visualize the pattern. Moreover, in a real-world example, the names of the players can be completely different than those in the pattern description.

Example 2-1. Decorator pattern theory code

```
1  using System;
2
3  class DecoratorPattern {
4
5      // Decorator Pattern          Judith Bishop  Dec 2006
6      // Shows two decorators and the output of various
7      // combinations of the decorators on the basic component
8
9      interface IComponent {
10         string Operation();
11     }
12
13     class Component : IComponent {
14         public string Operation () {
15             return "I am walking ";
16         }
17     }
18
19     class DecoratorA : IComponent {
20         IComponent component;
21
22         public DecoratorA (IComponent c) {
23             component = c;
24         }
25
26         public string Operation() {
27             string s = component.Operation();
28             s += "and listening to Classic FM ";
29             return s;
30         }
31     }
32
33     class DecoratorB : IComponent {
34         IComponent component;
35         public string addedState = "past the Coffee Shop ";
36
37         public DecoratorB (IComponent c) {
38             component = c;
39         }
40
41         public string Operation () {
42             string s = component.Operation ();
43             s += "to school ";
44             return s;
45         }
46
47         public string AddedBehavior() {
48             return "and I bought a cappuccino ";
49         }
50     }
51
52     class Client {
53
54         static void Display(string s, IComponent c) {
```

Example 2-1. Decorator pattern theory code (continued)

```
55     Console.WriteLine(s+ c.Operation());
56     }
57
58     static void Main() {
59         Console.WriteLine("Decorator Pattern\n");
60
61         IComponent component = new Component();
62         Display("1. Basic component: ", component);
63         Display("2. A-decorated : ", new DecoratorA(component));
64         Display("3. B-decorated : ", new DecoratorB(component));
65         Display("4. B-A-decorated : ", new DecoratorB(
66             new DecoratorA(component)));
67         // Explicit DecoratorB
68         DecoratorB b = new DecoratorB(new Component());
69         Display("5. A-B-decorated : ", new DecoratorA(b));
70         // Invoking its added state and added behavior
71         Console.WriteLine("\t\t\t"+b.addedState + b.AddedBehavior());
72     }
73 }
74 }
75 /* Output
76 Decorator Pattern
77
78 1. Basic component: I am walking
79 2. A-decorated : I am walking and listening to Classic FM
80 3. B-decorated : I am walking to school
81 4. B-A-decorated : I am walking and listening to Classic FM to school
82 5. A-B-decorated : I am walking to school and listening to Classic FM
83     past the Coffee Shop and I bought a cappuccino
84 */
```

The example starts off with the `IComponent` interface and a simple `Component` class that implements it (lines 9–17). There are two decorators that also implement the interface; each of them includes a declaration of an `IComponent`, which is the object it will decorate. `DecoratorA` (lines 19–31) is fairly plain and simply implements the `Operation` by calling it on the component it has stored, then adding something to the string it returns (line 28). `DecoratorB` (lines 33–50) is more elaborate. It also implements the `Operation` in its own way, but it offers some public `addedState` (line 35) and `addedBehavior` (lines 47–49) as well. In both implemented operations, the component's `Operation` method is called first, but this is not a requirement of the pattern; it merely makes for more readable output in this example.

The `Client` class is responsible for creating components and decorators in various configurations and displaying the result of calling the `Operation` in each case. Cases 2 and 3 (lines 63–64) decorate the basic component in different ways, as shown in the output on lines 79–80. Cases 4 and 5 apply two decorators, a `B` and an `A`, in different orders. In cases 2–4, the decorator objects are instantiated and used immediately, then discarded. In case 5, we create a `DecoratorB` object and keep this instance in a variable of the same type (instead of `IComponent`), so we can invoke the new behavior:

```

DecoratorB b = new DecoratorB(new Component());
Display("5. A-B-decorated : ", new DecoratorA(b));
// Invoking its added state and added behavior
Console.WriteLine("\t\t\t"+b.addedState + b.AddedBehavior());

```

**5. A-B-decorated : I am walking to school and listening to Classic FM
past the Coffee Shop and I bought a cappuccino**

There are three objects here: the explicitly declared `b` and the implicitly declared `DecoratorA` and `Component` objects. The `Display` method receives the `DecoratorA` object and invokes its `Operation` method (line 55). This takes it to the `DecoratorB` object. On line 42, the object that `DecoratorB` was composed with has its `Operation` invoked, and the basic string “I am walking” was returned (line 15). Continuing on, `DecoratorB` adds the bit about walking to school (line 43), and then `DecoratorA` adds “listening to Classic FM” (line 28). This completes the call to `Display`. As can be seen on line 82, the result is the opposite of line 81 because the decorators were composed in a different order.

However, that is not all. Decorators can define new behavior, which can be invoked explicitly. We do this on line 71, which results in the second line of output (line 83) about buying a cappuccino.

That, in a nutshell, is how decorators work. Decorators do not need any advanced language features; they rely on object aggregation and interface implementation. Now, let’s consider a real-world example.

Example: Photo Decorator

In this section, we’ll consider how to implement a photo decorator system, as described in the “Illustration” section, earlier. To emphasize that the original component was written previously and is not available for altering, we’ll place it in a separate namespace called `Given`:

```

using System.Windows.Forms;

namespace Given {
    // The original Photo class
    public class Photo : Form {
        Image image;

        public Photo () {
            image = new Bitmap("jug.jpg");
            this.Text = "Lemonade";
            this.Paint += new PaintEventHandler(Drawer);
        }

        public virtual void Drawer(Object source, PaintEventArgs e) {
            e.Graphics.DrawImage(image,30,20);
        }
    }
}

```

Photo inherits from System.Windows.Forms so that it can be displayed. The constructor sets up the Drawer method as the target of the PaintEventHandler that will be called when the Form is activated by a Main method. In C#, a call to Application.Run will start up a window in this way, so a basic client can look like this:

```
static void Main () {
    // Application.Run acts as a simple client
    Application.Run(new Photo());
}
```

Now, without altering anything in the Photo class, we can start adding decorators. The first draws a blue border around the photo (we'll assume the size is known to keep things simple):

```
// This simple border decorator adds a colored border of fixed size
class BorderedPhoto : Photo {
    Photo photo;
    Color color;
    public BorderedPhoto (Photo p, Color c) {
        photo = p;
        color=c;
    }

    public override void Drawer(Object source, PaintEventArgs e) {
        photo.Drawer(source, e);
        e.Graphics.DrawRectangle(new Pen(color, 10),25,15,215,225);
    }
}
```

Notice that this code deviates from the pattern laid out in Figure 2-2, in that there is no IComponent interface. This is perfectly acceptable; the decorators can inherit directly from the component and maintain an object of that class as well. We see now that Drawer is the operation that is to be called from one decorator to the next. However, this code does rely on the original Component declaring Drawer as virtual. If this is not the case, and we cannot go in and change the Component class, an interface is necessary. This arrangement is shown in Example 2-1.

The Tag decorator follows a similar form, so we can put together the following composition:

```
// Compose a photo with two tags and a blue border
foodTag = new Tag (photo, "Food", 1);
colorTag = new Tag (foodTag, "Yellow", 2);
composition = new BorderedPhoto(colorTag, Color.Blue);
Application.Run(composition);
```

The result of this sequence of decorations is shown in Figure 2-1(b). Notice that Decorators can be instantiated with different parameters to give different effects: the foodTag object has a second parameter of "Food", and the colorTag's second parameter is "Yellow".

Finally, let's examine the added state and behavior that is in the Tags. A Tag class declares a static array and counter that stores the tag names as they come in. Any tag decorator object can then call the ListTags method to show all the tags currently in play. The full program is given in Example 2-2.

Example 2-2. Decorator pattern example code—Photo Decorator

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
using System.Collections.Generic;
using Given;

// Decorator Pattern Example      Judith Bishop Aug 2007
// Draws a single photograph in a window of fixed size
// Has decorators that are BorderedPhotos and TaggedPhotos
// that can be composed and added in different combinations

namespace Given {

    // The original Photo class
    public class Photo : Form {
        Image image;
        public Photo () {
            image = new Bitmap("jug.jpg");
            this.Text = "Lemonade";
            this.Paint += new PaintEventHandler(Drawer);
        }

        public virtual void Drawer(Object source, PaintEventArgs e) {
            e.Graphics.DrawImage(image,30,20);
        }
    }

    class DecoratorPatternExample {

        // This simple BorderedPhoto decorator adds a colored border of fixed size
        class BorderedPhoto : Photo {
            Photo photo;
            Color color;

            public BorderedPhoto (Photo p, Color c) {
                photo = p;
                color=c;
            }

            public override void Drawer(Object source, PaintEventArgs e) {
                photo.Drawer(source, e);
                e.Graphics.DrawRectangle(new Pen(color, 10),25,15,215,225);
            }
        }
    }
}
```

Example 2-2. Decorator pattern example code—Photo Decorator (continued)

```
// The TaggedPhoto decorator keeps track of the tag number which gives it
// a specific place to be written
```

```
class TaggedPhoto : Photo {
    Photo photo;
    string tag;
    int number;
    static int count;
    List <string> tags = new List <string> ();

    public TaggedPhoto(Photo p, string t) {
        photo = p;
        tag = t;
        tags.Add(t);
        number = ++count;
    }

    public override void Drawer(Object source, PaintEventArgs e) {
        photo.Drawer(source,e);
        e.Graphics.DrawString(tag,
            new Font("Arial", 16),
            new SolidBrush(Color.Black),
            new PointF(80,100+number*20));
    }

    public string ListTaggedPhotos() {
        string s = "Tags are: ";
        foreach (string t in tags) s +=t+" ";
        return s;
    }
}

static void Main () {
    // Application.Run acts as a simple client
    Photo photo;
    TaggedPhoto foodTaggedPhoto, colorTaggedPhoto, tag;
    BorderedPhoto composition;

    // Compose a photo with two TaggedPhotos and a blue BorderedPhoto
    photo = new Photo();
    Application.Run(photo);
    foodTaggedPhoto = new TaggedPhoto (photo,"Food");
    colorTaggedPhoto = new TaggedPhoto (foodTaggedPhoto,"Yellow");
    composition = new BorderedPhoto(colorTaggedPhoto, Color.Blue);
    Application.Run(composition);
    Console.WriteLine(colorTaggedPhoto.ListTaggedPhotos());

    // Compose a photo with one TaggedPhoto and a yellow BorderedPhoto
    photo = new Photo();
    tag = new TaggedPhoto (photo,"Jug");
    composition = new BorderedPhoto(tag, Color.Yellow);
    Application.Run(composition);
}
```

Example 2-2. Decorator pattern example code—Photo Decorator (continued)

```
        Console.WriteLine(tag.ListTaggedPhotos());
    }
}
/* Output
TaggedPhotos are: Food Yellow
TaggedPhotos are: Food Yellow Jug
*/
```

An important point about the Decorator pattern is that it is based around new *objects* being created with their own sets of operations. Some of these might be inherited, but only down one level. For example, when implementing the Photo Decorator program, we could try to alter properties of the Windows Form class, such as Height and Width, from within a decorator object. For the component itself and for a first-level decorator, this would work. But as soon as there was a second level of decorators, the changes would not get through. The reason is evident from the object diagram in Figure 2-3: the first decorator holds a reference to the actual Windows Form object, but the second-level decorator does not. For this kind of manipulation a different pattern, such as Strategy (discussed in Chapter 7), would be more fitting.

In the example of the Photo Decorator program, we did not actually add any behavior; we only overrode it. A Client implemented as the Application.Run harness uses the event model to invoke the PaintEventHandler implicitly. It does not give an opportunity for calling other methods explicitly. Of course, other events (such as MouseMove and OnClick) can be programmed, and then the other behaviors in the decorators will make sense (see the “Exercises” section later).

Use

Here are four ways the Decorator pattern is used in the real world:

- As our small example illustrated, the Decorator pattern fits well in the graphics world. It is equally at home with video and sound; for instance, video streaming can be compressed at different rates, and sound can be input to a simultaneous translation service.
- At a more mundane level, decorators abound in the I/O APIs of C#. Consider the following hierarchy:

```
System.IO.Stream
    System.IO.BufferedStream
    System.IO.FileStream
    System.IO.MemoryStream
    System.Net.Sockets.NetworkStream
    System.Security.Cryptography.CryptoStream
```

The subclasses decorate Stream because they inherit from it, and they also contain an instance of a Stream that is set up when an object is constructed. Many of their properties and methods relate to this instance.

- In today's world of mobile devices, web browsers and other mobile applications thrive on the Decorator pattern. They can create display objects suitable for smaller screens that include scroll bars and exclude banners that would be standard on desktop display browsers, for example.
- The Decorator pattern is so useful that there are now actual Decorator classes in .NET 3.0. The one in `System.Windows.Controls` “provides a base class for elements that apply effects onto or around a single child element, such as `Border` or `Viewbox`.”

The following table summarizes when, in general, to use the Decorator pattern.

Use the Decorator pattern when...

You have:

- An existing component class that may be unavailable for subclassing.

You want to:

- Attach additional state or behavior to an object dynamically.
- Make changes to some objects in a class without affecting others.
- Avoid subclassing because too many classes could result.

But consider using instead:

- The Adapter pattern, which sets up an interface between different classes.
- The Composite pattern, which aggregates an object without also inheriting its interface.
- The Proxy pattern, which specifically controls access to objects.
- The Strategy pattern, which changes the original object rather than wrapping it.

Exercises

1. Assume that the `Photo` class was written with `Drawer` as a plain (not virtual) method and it cannot be altered. Reconstruct Example 2-2 so that it works under this constraint. (Hint: use an interface as in the theory example.)
2. Add to the `Photo` Decorator system a second type of component called `Hominid`. Use the drawing methods for ovals and lines to draw an approximation of a person. Then, use the same two decorators—`Tag` and `Border`—to decorate `Hominid` objects.
3. Add other event handlers to the constructors of the decorators, together with additional behavior. For example, an `OnClick` event could cause a tag to appear, rather than having it appear automatically.
4. Decorate the `Console` class so that `Write` and `WriteLine` methods are trapped and the output is reformatted for lines of a given size, avoiding unsightly wrap-arounds. Test your decorator with the program in Example 2-1.
5. Write a program that decorates the `Stream` class and shows how much of a file has been read in, using a trackbar.