**Chương 7: Mẫu Decorator**

Mẫu Decorator cung cấp cho chúng ta một phương pháp để thay đổi hành vi của đối tượng cụ thể mà không cần phải tạo ra một lớp mới kế thừa từ lớp của đối tượng đó. Giả sử chúng ta có một chương trình sử dụng tám đối tượng, nhưng ba trong số chúng cần một tính năng bổ sung. Chúng ta có thể tạo ra một lớp dẫn xuất cho từng đối tượng, và trong nhiều trường hợp đây sẽ là một giải pháp hoàn toàn chấp nhận được. Tuy nhiên, nếu mỗi một trong ba đối tượng yêu cầu các tính năng *khác nhau*, điều này sẽ có nghĩa là phải tạo thêm ba các lớp dẫn xuất khác nhau. Hơn thế nữa, nếu một trong các lớp này có các tính năng mà *cả hai* lớp khác có, ta sẽ bắt đầu gặp phải một phức tạp bao gồm sự khó hiểu và không cần thiết.

Ví dụ, giả sử chúng ta muốn vẽ một đường biên đặc biệt xung quanh một số buttton trong thanh công cụ. Nếu chúng ta tạo ra một lớp dẫn xuất Button mới, đồng nghĩa với việc tất cả các đối tượng Button trong lớp mới này sẽ luôn luôn có chung một kiểu đường biên đặc biệt này, ngay cả khi đó không phải là những gì chúng ta muốn.

Thay vào đó, chúng ta tạo ra một lớp Decorator để trang trí cho các button mà ta muốn. Sau đó chúng ta dẫn xuất một số lượng bất kỳ các Decorator cụ thể mà ta cần từ các lớp Decorator chính ở trên, mỗi trong số đó chúng sẽ thực hiện một công việc trang trí cụ thể. Để trang trí một nút, Decorator đó phải là một đối tượng được dẫn xuất từ môi trường trực quan để nó có thể nhận được lời gọi các phương thức vẽ và chuyển tiếp các lời gọi các phương thức vẽ khác cho các đối tượng đang được trang trí. Đây là một trường hợp mà việc chứa một đối tượng được ưa chuộng hơn thừa kế một đối tượng. Các Decorator là một đối tượng đồ họa, nhưng chúng lại chứa các đối tượng mà chúng đang trang trí. Điều này giúp ngăn chặn một số lời gọi phương thức đồ họa, thực hiện một số tính toán bổ sung, và chuyển chúng sang các đối tượng cơ bản đang được chúng trang trí.

### **Trang trí một CoolButton**

Các ứng dụng Windows gần đây như Internet Explorer và Netscape Navigator có một hàng các nút phẳng, không viền và tự làm nổi bật bản thân chúng với các đường viền khi bạn di chuyển chuột qua chúng. Một số lập trình viên Windows gọi toolbar này là CoolBar và các button là coolButton. Không có các các control nào tương tự như thế trong C# nên chúng ta có thể tạo ra các hành vi như vậy bằng cách trang trí một Panel và dùng đối tượng panel này chứa trong đó một button. Trong trường hợp này, chúng ta trang trí nó bằng cách vẽ đường viền đen và trắng để làm nổi bật các button (khi di chuyển chuột qua) hoặc các đường màu xám để loại bỏ các đường biên của button.

Hãy xem xét làm thế nào để tạo ra Decorator này. Mẫu Thiết kế cho thấy rằng Decorator nên được dẫn xuất từ một lớp thành phần trực quan nói chung, và sau đó mỗi thông điệp cho các button thực tế nên được chuyển tiếp từ Decorator chứa button đó. Đây không phải là tất cả những thực tế trong C#, nhưng nếu chúng ta sử dụng các container như là các Decorator, tất cả các sự kiện được chuyển tiếp đến control được chứa bên trong container.

Mẫu thiết kế xa hơn nữa còn đề nghị rằng các class như Decorator nên là các lớp trừu tượng và ta cần dẫn xuất tất cả các concrete Decorator của chúng ta từ các lớp trừu tượng này. Trong việc hiện thực mẫu, chúng ta định nghĩa một giao diện Decorator sẽ nhận được các sự kiện vẽ và các sự kiện về hành vi của chuột mà chúng ta cần nắm bắt.

interface Decorator

{

void MouseMove(object sender, MouseEventArgs e);

void MouseEnter(object sender, EventArgs e);

void MouseLeave(object sender, EventArgs e);

void Paint(object sender, PaintEventArgs e);

}

Trong hiện thực thực tế của chúng ta, chúng ta có thể dẫn xuất được một CoolDecorator từ một lớp Panel và có nó trở thành container lưu trữ button mà chúng ta sẽ trang trí.

Bây giờ, chúng ta hãy xem làm thế nào chúng ta có thể hiện thực một CoolButton. Tất cả chúng ta cần phải làm là vẽ một đường viện màu trắng và đen  
xung quanh diện tích button khi nó nhận được focus (khi rê chuột, hay các sự kiện tương ứng) và vẽ các đường màu xám khi không có các yếu tố trên. Khi một sự kiện MouseMove được phát hiện trên các button, sự kiện vẽ tiếp theo nên vẽ các viền với màu đã chọn, và khi con chuột rời khỏi button, sự kiện vẽ tiếp theo nên vẽ đường viền màu xám. Chúng ta làm điều này bằng cách thiết lập một cờ mouse\_over và sau đó repaint lại bằng cách gọi phương thức Refresh.

public void MouseMove(object sender, MouseEventArgs e){

mouse\_over = true;

}

public void MouseEnter(object sender, EventArgs e){

mouse\_over = true;

this.Refresh ();

}

public void MouseLeave(object sender, EventArgs e){

mouse\_over = false;

this.Refresh ();

}

Dưới đây là sự kiện vẽ thật sự

public virtual void paint(object sender, PaintEventArgs e)

{

//draw over button to change its outline

Graphics g = e.Graphics;

const int d = 1;

//draw over everything in gray first

g.DrawRectangle(gPen, 0, 0, x2 - 1, y2 - 1);

//draw black and white boundaries

//if the mouse is over

if (mouse\_over)

{

g.DrawLine(bPen, 0, 0, x2 - d, 0);

g.DrawLine(bPen, 0, 0, 0, y2 - 1);

g.DrawLine(wPen, 0, y2 - d, x2 - d, y2 - d);

g.DrawLine(wPen, x2 - d, 0, x2 - d, y2 - d);

}

}

## **Điều khiển các Event trong một Decorator**

Khi chúng ta xây dựng một Decorator thực tế có chứa các phương thức cho việc xử lí chuột và vẽ mà chúng ta chỉ ra, chúng ta phải kết nối các hệ thống xử lý sự kiện (event handling) cho các phương thức này. Chúng ta làm điều này trong constructor của Decorator bằng cách tạo ra một lớp EventHandler cho sự kiện khi chuột ở trên button và một MouseEventHandler cho việc di chuyển và rời các sự kiện. Điều quan trọng là cần lưu ý rằng các sự kiện chúng ta đang muốn bắt là các sự kiện vào button được chứa trong container, chứ không phải trên Panel xung quanh nó. Vì vậy, đối tượng mà chúng ta thêm các handler trên là của riêng button.

public void CoolDecorator(Control c)

{

control = c; //copy in control

//mouse over, enter handler

EventHandler evh = new EventHandler(MouseEnter);

c.MouseHover += evh;

c.MouseEnter += evh;

//mouse move handler

c.MouseMove += new MouseEventHandler(MouseMove);

c.MouseLeave += new EventHandler(MouseLeave);

Tương tự như vậy, chúng ta tạo ra một PaintEventHandler cho sự kiện vẽ.

//paint handler catches button's paint

c.Paint += new PaintEventHandler(paint);

}

***Cải thiện Layout***

Nếu bạn tạo một form Windows chứa các button, các GUI designer tự động tạo code để thêm các Control đó vào một mảng chứa các Control cho form đó. Chúng ta muốn thay đổi điều này bằng cách thêm button vào mảng Control cho panel mới, thêm panel để các mảng Control cho form, và loại bỏ các button có trong mảng đó. Dưới đây là đoạn code để thêm panel và loại bỏ các button trong phương thức khởi tạo Form.

//add outside decorator to the layout

//and remove the button from the layout

this.Controls.AddRange(new System.Windows.Forms.Control[] { cdec });

this.Controls.Remove(btButtonA);

Đây là đoạn code để thêm một button vào Decorator panel

public CoolDecorator(Control c)

{

contl = c; //copy in control

//add button to controls contained in panel

this.Controls.AddRange(new Control[] { contl });

}

***Kích thước và Vị trí Control***

Khi chúng ta trang trí các button bằng cách đặt nó trong một Panel, chúng ta cần phải thay đổi tọa độ và kích thước để các Panel có kích thước và tọa độ của button và button có một vị trí (0, 0) trong bảng điều khiển. Điều này cũng xảy ra trong các nhà xây dựng CoolDecorator

this.Location = p;

contl.Location = new Point(0, 0);

this.Name = "deco" + contl.Name;

this.Size = contl.Size;

x1 = c.Location.X - 1;

y1 = c.Location.Y - 1;

x2 = c.Size.Width;

y2 = c.Size.Height;

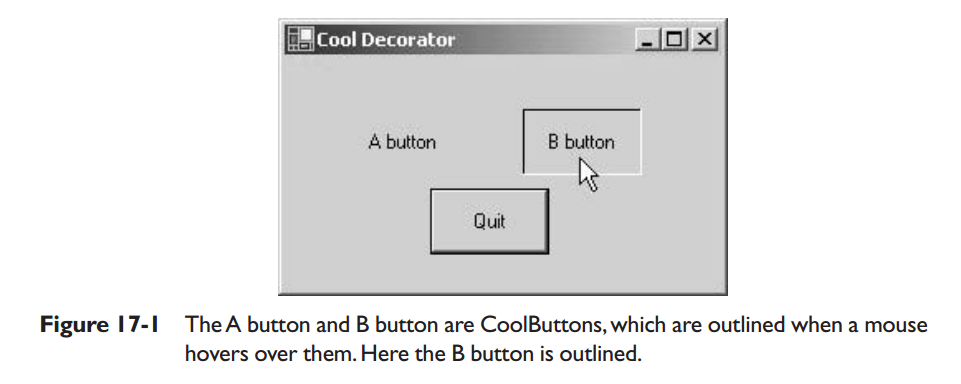
Chúng ta cũng tạo ra thể hiện của Pens để sử dụng trong các phương thức vẽ trong constructor này.

//create the overwrite pens

gPen = new Pen(c.BackColor, 2); //gray pen overwrites borders

bPen = new Pen(Color.Black, 1);

wPen = new Pen(Color.White, 1);



## **Kết hợp nhiều Decorator**

Bây giờ chúng ta thấy làm thế nào một Decorator duy nhất hoạt động, vậy còn về về đa Decorator thì sao? Có thể là chúng ta muốn để trang trí CoolButtons của chúng ta với một phong cách mới – chẳng hạn, một đường chéo màu đỏ. Điều này chỉ chút phức tạp hơn một chút bởi vì chúng ta chỉ cần phải bao CoolDecorator bên trong thêm một Decorator khác để nhiều kiểu trang trí diễn ra hơn. Sự thay đổi thực sự duy nhất là chúng ta không cần thể hiện của panel mà chúng ta đang gói mà còn là đối tượng trung tâm (ở đây là một button) cần được trang trí, khi chúng ta phải attach các phương thức vẽ của chúng ta để vào pháp thức vẽ của đối tượng trung tâm.

Vì vậy, chúng ta cần tạo một constructor cho Decorator của chúng ta có cả panel được đóng gói và button như các control khác.

public class CoolDecorator :Panel, Decorator {

protected Control contl;

protected Pen bPen, wPen, gPen;

private bool mouse\_over;

protected float x1, y1, x2, y2;

//----------------------------------

public CoolDecorator(Control c, Control baseC)

{

//the first control is the one laid out

//the base control is the one whose paint method we extend

//this allows for nesting of decorators

contl = c;

this.Controls.AddRange(new Control[] { contl });

Sau đó, chúng ta thêm các event handler, xử lý sự kiện vẽ phải được gắn liền với sự control cha.

//paint handler catches button's paint

baseC.Paint += new PaintEventHandler(paint);

Chúng ta tạo nên một phương thức vẽ virtual để có thể overrie nó như chúng ta thấy sau đây

public virtual void paint(object sender, PaintEventArgs e){

//draw over button to change its outline

Graphics g = e.Graphics;

Hoá ra rằng cách dễ nhất để viết SlashDecorator, cái sẽ vẽ ra đường chéo màu đỏ cho button, là trực tiếp dẫn xuất từ CoolDecorator. Chúng ta có thể tái sử dụng tất cả các phương thức của CoolDecorator, đồng thời chỉ cần extend phương thức vẽ từ CoolDecorator và tiết kiệm rất nhiều công sức.

public class SlashDecorator : CoolDecorator

{

private Pen rPen;

//----------------

public SlashDeco(Control c, Control bc)

: base(c, bc)

{

rPen = new Pen(Color.Red, 2);

}

//----------------

public override void paint(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

x1 = 0; y1 = 0;

x2 = this.Size.Width;

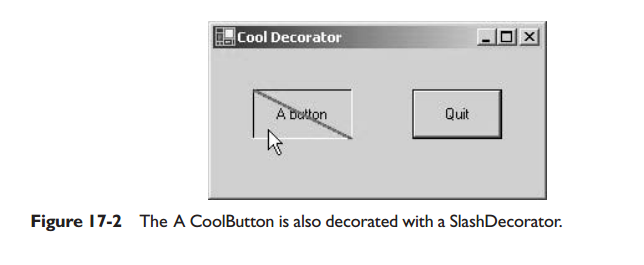
y2 = this.Size.Height;

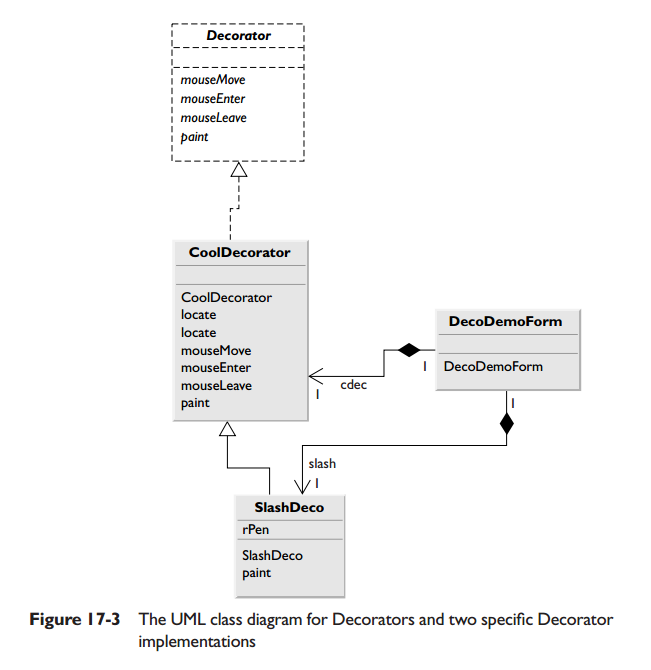
g.DrawLine(rPen, x1, y1, x2, y2);

}

}

Cuối cùng chúng ta có được một chương trình cuối cùng hiển thị hai button, như được hiển thị trong hình 17-2. Sơ đồ lớp được thể hiện trong hình 17-3.





## **Decorator không trực quan**

Trang trí, tất nhiên, không giới hạn đối tượng mà tăng cường các lớp học trực quan. Bạn có thể thêm hoặc sửa đổi các phương thức của bất kỳ đối tượng theo một cách tương tự. Trong thực tế, các đối tượng phi hình ảnh có thể được dễ dàng hơn để trang trí vì có thể có ít hơn phương thức để nắm bắt và chuyển tiếp. Bất cứ khi nào bạn đặt một thể hiện của một lớp bên trong một lớp khác và có các lớp bên ngoài hoạt động trên đó, bạn cơ bản là "trang trí" lớp bên trong. Đây là một trong những công cụ phổ biến nhất cho các chương trình có sẵn trong Visual Studio.NET

## **Decorators, Adapters, và Composites**

Như đã nêu trong mẫu thiết kế, có một sự tương đồng quan trọng trong những class bạn có thể đã được công nhận. Adapters cũng dường như để "trang trí" một class hiện có. Tuy nhiên, chức năng của chúng là thay đổi giao diện của một hoặc nhiều class để một trong số đó thuận tiện hơn cho một chương trình cụ thể. Các Decorator thêm các phương thức cho các trường hợp đặc biệt của các class chứ không phải là tất cả trong số chúng. Bạn cũng có thể tưởng tượng rằng đó là một composite của một đối tượng duy nhất về bản chất là một Decorator. Một lần nữa, tuy nhiên, mục đích là khác nhau.

## **Hệ quả của việc các mẫu Decorator**

Các mẫu Decorator cung cấp một cách linh hoạt hơn để thêm trách nhiệm cho một class hơn là bằng cách sử dụng kế thừa, vì nó có thể thêm những trách nhiệm cho các instance được lựa chọn của lớp. Nó cũng cho phép bạn tùy chỉnh một class mà không cần tạo lớp con mới trong hệ thống phân cấp thừa kế. Mẫu thiết kế chỉ ra hai nhược điểm của mẫu Decorator. Một là một Decorator và thành phần kèm theo của nó không giống nhau. Vì vậy, kiểm thử đối với loại của đối tượng sẽ thất bại. Thứ hai là các Decorator có thể dẫn đến một hệ thống với "rất nhiều đối tượng nhỏ" với tất cả đều giống nhau để các lập trình viên cố gắng để bảo trì mã. Điều này có thể là một vấn đề nhức đầu trong việc bảo trì. Decorator và Façade gợi lên hình ảnh tương tự trong việc xây dựng kiến trúc, nhưng trong thuật ngữ mẫu thiết kế, Façade là một cách để che giấu một hệ thống phức tạp bên trong một giao diện đơn giản, trong khi đó Decorator thêm chức năng bằng cách bọc một class. Chúng ta sẽ bàn luận về Façade trong chương tiếp theo.

## **Suy nghĩ về các câu hỏi**

1: Khi ai đó nhập vào một giá trị không chính xác trong một ô của một lưới, có thể bạn muốn thay đổi màu sắc của hàng để chỉ ra các vấn đề. Đề nghị làm thế nào bạn có thể sử dụng một Decorator.

2: Một quỹ tương hỗ là một bộ sưu tập của các cổ phiếu. Mỗi một bao gồm một mảng, hoặc Collection, giá theo thời gian. Bạn có thể tìm cách thiết kế một Decorator có thể được sử dụng để tạo ra một báo cáo về hiệu suất cổ phiếu cho mỗi cổ phiếu và cho toàn bộ quỹ?