Alex Lang

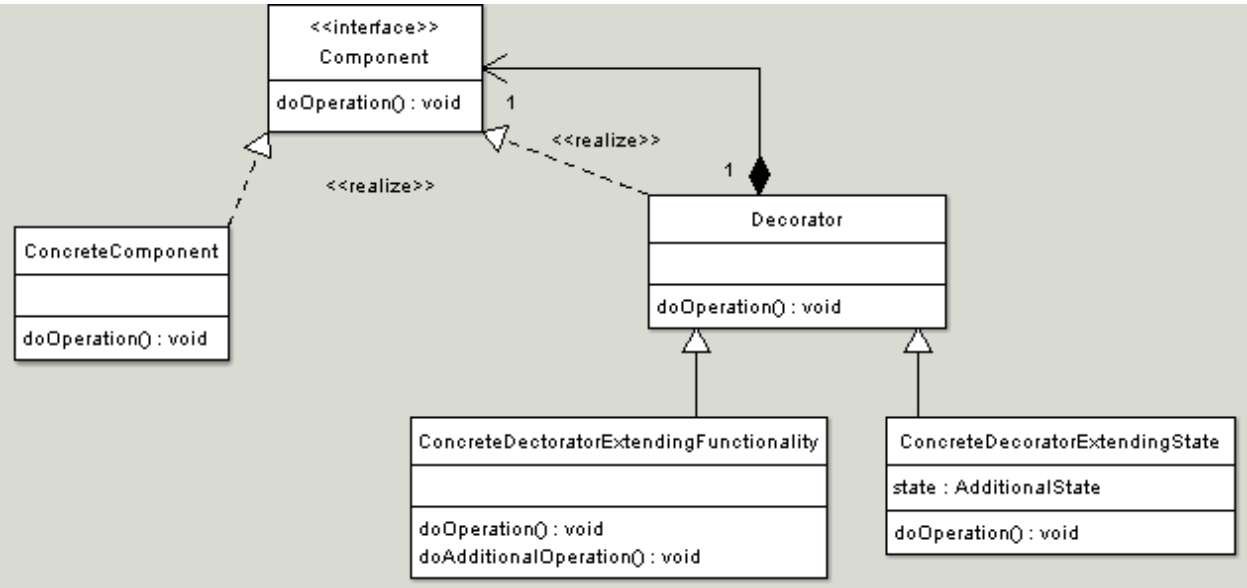
The Decorator Pattern

Design Patterns

11/17/16

For this assignment, we were told to do the Decorator Pattern. The purpose of the decorator pattern is to add additional responsibilities dynamically to an object. This pattern makes very good use of polymorphism, and makes making changes to object much easier.

Here is the UML diagram from OODesign on what the Decorator Pattern consists of and its implementation.



For my demonstration of the Decorator Pattern, I decided to make a simple application that lets the user decorate a shape, in this case, a circle and a rectangle. The user selects from a couple of radio buttons which shape they would like, and then another selection of radio buttons for the color of the selected shape.

Starting off I created the component class for this pattern, which I named shape. Since according to the diagram, everything inherits from the component directly or indirectly, I started with this first.

```
public abstract class Shape //Component
    {
        public abstract void drawShape(Form1 f, Pen colorPen);
    }
```

Here I created drawShape. This is the only method needed here because regardless of the state of the shape, whether it is decorated or not, the app still needs to draw it. The arguments for this method are Form1 and a Pen object. The form is passed in so the application knows where to draw, and the pen is passed in so the shape can have its color and thickness.

Next I created my Concrete Component classes. For this app, I went against the UML diagram a little, since the UML only says there is one Concrete Component class. I felt this really wasn't an issue, since this was necessary to properly display this pattern for my needs. Both shapes were objects that can have additional responsibilities added which is the definition for Concrete Components, so I went ahead with it.

```
public class Circle : Shape //Concrete Component
    {
        Pen colorPen = new Pen(Color.Black, 5); //Default value


        public override void drawShape(Form1 f, Pen colorPen)
        {
            Graphics graphics = f.CreateGraphics();
            f.CreateGraphics().Clear(Form1.ActiveForm.BackColor);
            graphics.DrawEllipse(colorPen, 680, 100, 80, 80);
        }


    }

public class Rectangle : Shape //Concrete Component
    {
        Pen colorPen = new Pen(Color.Black, 5); //default value


        public override void drawShape(Form1 f, Pen colorPen)
        {
            Graphics graphics = f.CreateGraphics();
            f.CreateGraphics().Clear(Form1.ActiveForm.BackColor);
            graphics.DrawRectangle(colorPen, 680, 100, 80, 80);
        }


    }
```

Both of these concrete component classes worth very similarly. The draw shape method here creates a graphics canvas on the main form, clears any existing graphics, then draws the respective shape on the canvas. The default value of the Pen color is black to try and prevent any null pointer errors.

Next, I created my abstract decorator class.

```
public abstract class Drawer : Shape //Decorator
    {
        Shape shape;
        Pen colorPen;
    }
```

Since drawer already inherits from shape, I didn't need to add too much to this class. The only similar things the decorator classes shared with one another are the drawShape method that already was inherited, and the fact that they use some abstract shape with some abstract color.

After that, I created my concrete decorator classes. There are all extremely similar, the only difference being the Pen color object they have and the constructor. Here are a couple of the four I created, the decorator classes being red, blue, purple and green.

```
public class Red : Drawer //ConcreteDecorator
    {

        Shape shape;

        public Red(Shape shape)
        {
            this.shape = shape;
        }

        public override void drawShape(Form1 f, Pen colorPen)
        {
            colorPen = new Pen(Color.Red, 5);
            shape.drawShape(f, colorPen);
        }


    }

public class Blue : Drawer //Concrete Decorator
    {
        Shape shape;

        public Blue(Shape shape)
        {
            this.shape = shape;
        }
```

```
        public override void drawShape(Form1 f, Pen colorPen)
        {
            colorPen = new Pen(Color.Blue, 5);
            shape.drawShape(f, colorPen);
        }

    }
```

These decorator classes here are what really define the decorator pattern. These classes are passed in an abstract shape, and when drawShape is called in this class, the abstract shape is drawn in that color.

Finally, I put everything together in the form.

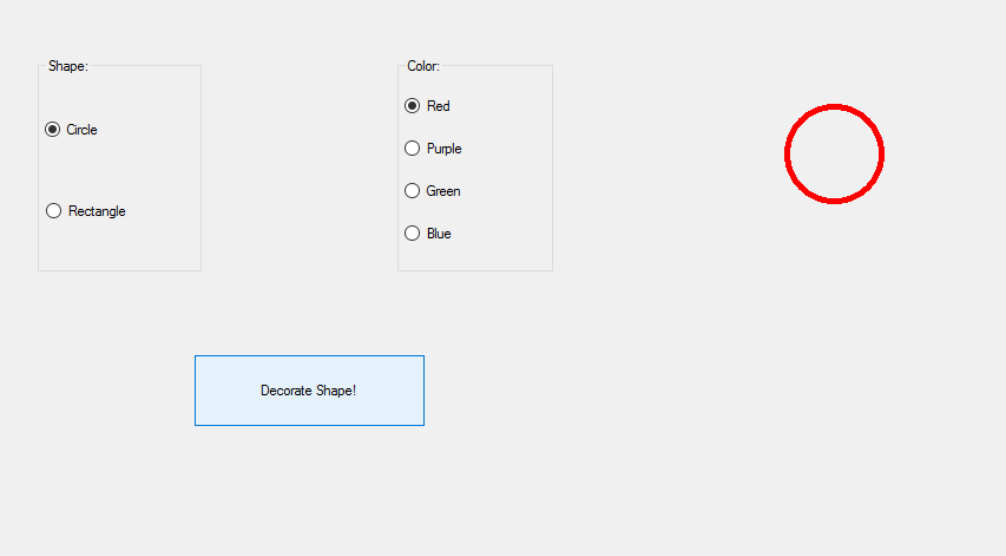
```
public partial class Form1 : Form
    {
        Shape shape;
        Pen pen;

        public Form1()
        {
            InitializeComponent();
            circleRBtn.Checked = true;
            redRBtn.Checked = true;
        }

        private void decorateBtn_Click(object sender, EventArgs e)
        {

            if (circleRBtn.Checked)
            {
                shape = new Circle();
            }

            else if (rectRBtn.Checked)
            {
                shape = new Rectangle();
            }

            if (redRBtn.Checked)
            {
                shape = new Red(shape);
                shape.drawShape(this, pen);
            }
            else if (purpleRBtn.Checked)
            {
                shape = new Purple(shape);
                shape.drawShape(this, pen);
            }
            else if (greenRBtn.Checked)
```

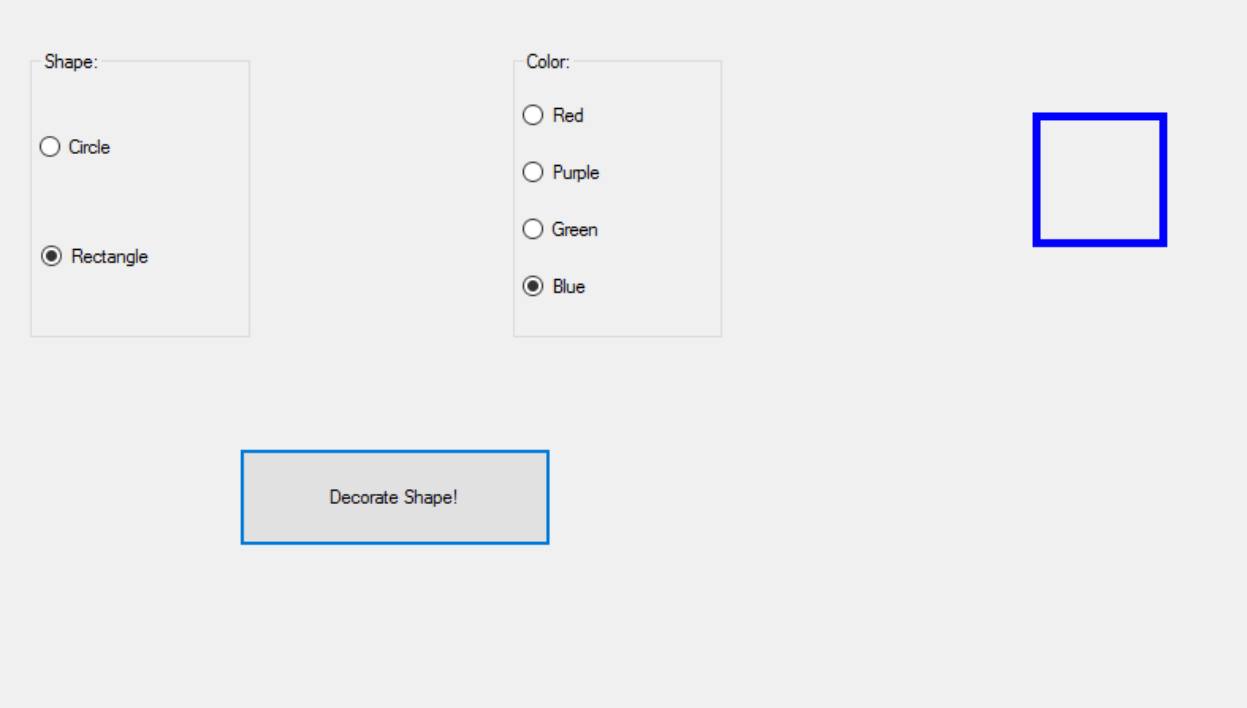
```
            {
                shape = new Green(shape);
                shape.drawShape(this, pen);
            }
            else if (blueRBtn.Checked)
            {
                shape = new Blue(shape);
                shape.drawShape(this, pen);
            }


        }
    }
```

To start off, I declared shape and pen objects and did some other initialization. The first couple radio buttons are what are defining our shape. If the circle radio button is selected, the shape will be a circle, if rectangle is selected the shape will be a rectangle.

The next four radio buttons are the decorators. If for example, the red radio button is selected, the shape becomes a new red shape, created from the shape passed in. The shape is then drawn based on that specific decoration.

Some screenshots of the app running:





Conclusion: I loved this pattern. When I first created this pattern, I thought I had an understanding of what it was, but I really didn't. I had a few classes that originally didn't really do much, they were more there to fit the UML diagram specifications. I was laying in bed and I really thought about this app, and then it suddenly came to me and I recreated most of it, and I think I did a much better job of what I did before. I can see how this pattern will be useful later down the road, and I really enjoyed all that I learned from creating this application.