The Decorator Pattern

Design Patterns

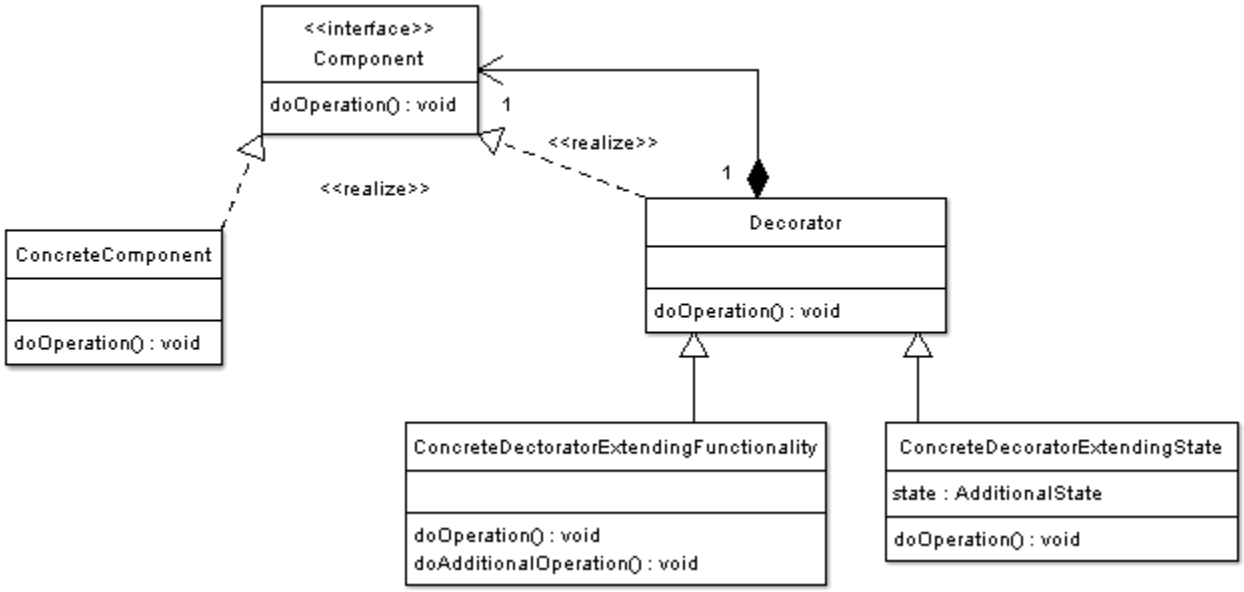
11/17/16

Cory Heitkamp

Introduction:

For this assignment, we were tasked with using the decorator pattern in an application. The decorator pattern is used to add to an object dynamically. My particular program is a calculator that does addition and subtraction, and you can decorate it with the ability to do multiplication as well.

UML Diagram: The diagram shows the most necessary components and functions used for the decorator pattern. It describes how the classes are connected, and some of the functions contained inside.



This table below summarizes the classes that I've created and used to make the decorator pattern.

Calculator	An abstract class with functions defined by the concrete calculator.
Concrete Calculator	Defines the add and subtract functions from the calculator class.
Decorator	Class with functions to be added to by the concrete class.
Concrete Decorator	Class that adds functionality to the display function.
Form	My windows form application

Narrative: For my application I decided to make a calculator. I began by writing the calculator class, which is my component. It has three abstract functions to be defined by the concrete class.

```
  public abstract int add(int x, int y);
      public abstract int substract(int x, int y);
      public abstract string display(int x, int y);
```

The Concrete Calculator class defines the functionality of the abstract methods.

```
        public override int add(int x, int y)
        {
            return x + y;
        }

        public override string display(int x, int y)
        {
            return "Add: "+x + "+" + y + "=" + add(x, y) + " \n" +
               "Subtract: " +x + "-" + y + "=" + substract(x, y);
        }

        public override int substract(int x, int y)
        {
            return x - y;
        }
```

The Decorator class inherits from the class, and uses an instantiation of the Calculator class. It uses this instantiation to provide definitions for the functions.

```
public class Decorator : ConcreteCalculator
    {
        protected Calculator math = new ConcreteCalculator();

        public override int add(int x, int y)
        {
            return math.add(x, y);
        }

        public override string display(int x, int y)
        {

            return math.display(x,y);
        }

        public override int substract(int x, int y)
        {
            return math.substract(x, y);
        }
```

I then created the Concrete decorator class, which adds functionality to the display function. It also adds a multiply function, which is used in the enhanced display function.

```
  public class ConcreteDecorator: Decorator
    {
        public int multiply(int x, int y)
        {
            return x * y;
        }
        public override string display(int x, int y)
        {
            return base.display(x, y) + "\n" +
                x + "times" + y + "=" + multiply(x, y);
```

```
        }
    }
```

Finally, I created the form itself. I wrote an instance of a Calculator, and of a decorator. I would then run the display function for the Calculator or for the decorator depending on which button was clicked.

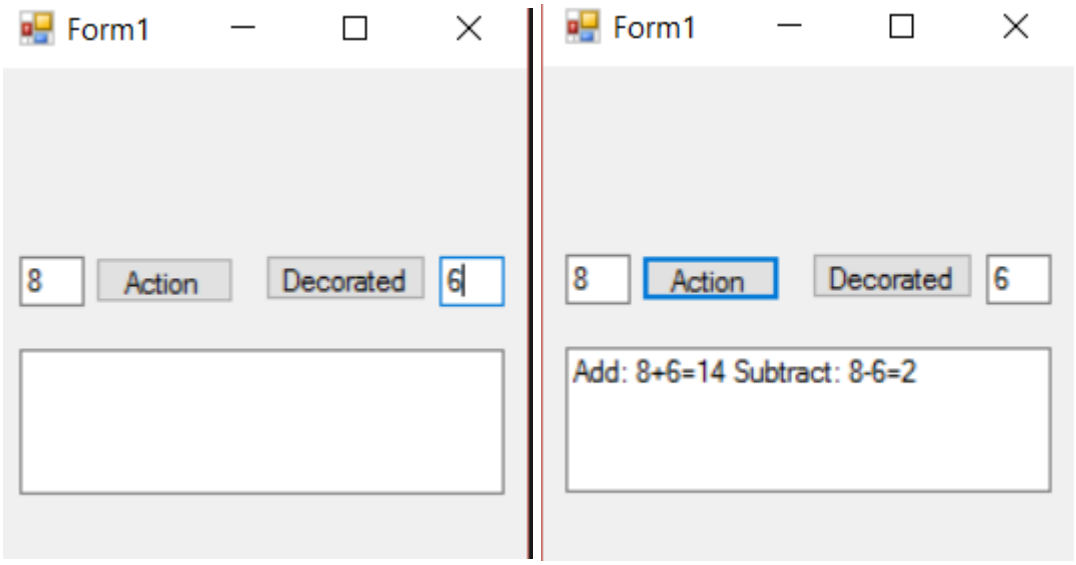
```
      Calculator maths = new ConcreteCalculator();
        Decorator decor = new ConcreteDecorator();
```

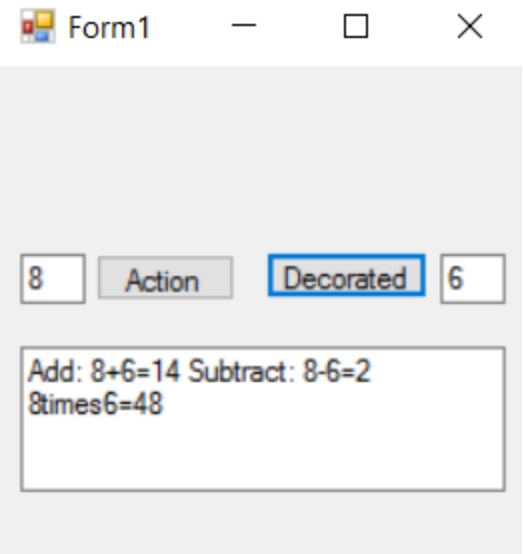
I would then run the display function for the Calculator or for the decorator depending on which button was clicked.

```
private void actionbutton_Click(object sender, EventArgs e)
        {
            displaytextBox.Text = maths.display(Convert.ToInt32( firsttextBox.Text),
Convert.ToInt32( secondtextBox.Text));
        }

  private void decorbutton1_Click(object sender, EventArgs e)
        {
            displaytextBox.Text = decor.display(Convert.ToInt32(firsttextBox.Text),
Convert.ToInt32(secondtextBox.Text));
        }
```

When ran, it looks like this:





Reflection: For the decorator pattern, it was easy to find a demo that would be a representation of the pattern. My only real concern was on the UML diagram where there was two concrete decorator classes, but upon further looking I found that there was a choice between adding implementation or a further state, so I added implementation. Overall, I think it went well.