Alex Lang

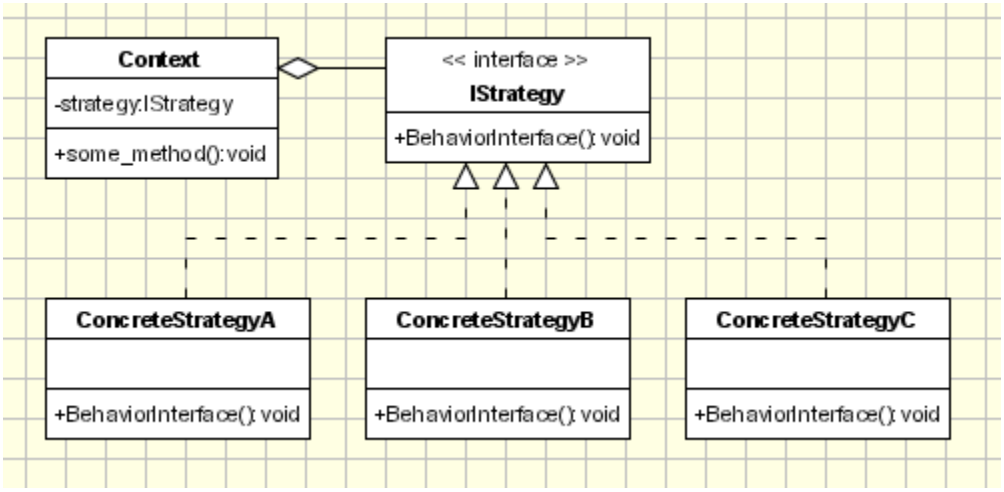
The Strategy Pattern

Design Patterns

10/20/16

For this assignment, we were told to do the Strategy Design Pattern. The Strategy Pattern defines a family of algorithms, encapsulating each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. What this means is that the pattern takes objects given to it, and does different operations to it, depending on the strategy used.

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



For my assignment, I chose to create a program that displays various strategies one can take when doing their work. There are four different strategy buttons for this program, getting the work done ASAP, early with breaks, procrastinating, and not doing the work altogether. When the button is clicked, text is returned to the display box and explains the results of each individual strategy is.

The first class I created was my abstract strategy class, HomeworkStrategy.

```
public abstract class HomeworkStrategy
    {
        public abstract string getStrategy();
    }
```

This class defines the getStrategy method that each strategy object will contain. They all get the strategy description, and return a string.

Next, I created all the classes to match each of the different methods for completing the homework.

```
public class BlowOff : HomeworkStrategy
    {
        public override string getStrategy()
        {
            return "Fun had in other activities:" + "\r\n" +
               "High, but the guilt of not completing assignments might eat you up later,
lowering fun." + "\r\n \r\n" +

               "Time taken:" + "\r\n" +
               "None." + "\r\n \r\n" +

               "Sleep:" + "\r\n" +
               "High." + "\r\n \r\n" +

               "Understanding of topic:" + "\r\n" +
               "None, you will fail any tests on subject. NOT suggested by any means.";
        }
    }
```

All the strategies used are extremely similar, the only differences being the string that they return. The BlowOff class for example, inherits from HomeworkStrategy and defines getStrategy(). The strategy is the string being returned that explains the pros and cons for the following strategy. The rest of the strategy classes are below.

```
public class Procrastinate : HomeworkStrategy
    {
        public override string getStrategy()
        {
            return "Fun had in other activities:" + "\r\n" +
                "High, until you realize you have to finally do your work. Work
anticipation lowers fun overrall." + "\r\n \r\n" +

                "Time taken:" + "\r\n" +
                "High, since you're already tired, doing the work and learning it becomes
very difficult. No concentration." + "\r\n \r\n" +

                "Sleep:" + "\r\n" +
                "Very little as a result of staying up all night doing it." + "\r\n \r\n"
+

                "Understanding of topic:" + "\r\n" +
                "Little to none since rushing to complete assignment.";
        }
    }

public class Breaks : HomeworkStrategy
    {
        public override string getStrategy()
        {
            return "Fun had in other activities:" + "\r\n" +
                "Medium-High, You take breaks periodically while doing work, still plenty
of time after assignment complete." + "\r\n \r\n" +

                "Time taken:" + "\r\n" +
                "Medium/High, depends on amount of breaks taken. Hard to go back to
assignment after starting." + "\r\n \r\n" +

                "Sleep:" + "\r\n" +
                "Average, since completing assignment early, just with a few breaks." +
"\r\n \r\n" +

                "Understanding of topic:" + "\r\n" +
                "High, breaks can be good to rest brain.";
        }
    }
```

```
public class DoneQuick : HomeworkStrategy
    {
        public override string getStrategy()
        {
            return "Fun had in other activities:" + "\r\n" +
               "Starts low while doing work, but after work is all out of the way you're
free to do whatever you want for rest of day." + "\r\n \r\n" +

               "Time taken:" + "\r\n" +
               "Low, you get it out of the way without distractions." + "\r\n \r\n" +

               "Sleep:" + "\r\n" +
               "High, since you don't have to worry about work, you can choose to sleep
whenever you wish." + "\r\n \r\n" +

               "Understanding of topic:" + "\r\n" +
               "Very high, some other studying is neccessary, but the work was put into
the assignment.";
        }
    }
```

The last class I created was the ConcreteStrategy class. What this class serves to do is just to tie all the strategies together, and make the code on the form cleaner.

```
public class ConcreteStrategy : HomeworkStrategy
    {
        private HomeworkStrategy hwStrat;

        public void setStrategy(HomeworkStrategy hwStrat)
        {
            this.hwStrat = hwStrat;
        }

        public override string getStrategy()
        {
            return hwStrat.getStrategy();
        }
    }
```

The ConcreteStrategy class, much like the other classes, inherits from HomeworkStrategy, and defines getStrategy. This class however, is passed in the homework strategy objects retrieved from the form and set as hwStrat. Next, the getStrategy uses the strategy of the method passed in, whether it be the procrastination strategy, the break strategy or some other. As stated before, this is done so the code can look cleaner on the main form, and each button is only calling on one object for methods directly.

```
    public partial class Form1 : Form
{

    ConcreteStrategy cs = new ConcreteStrategy();

    public Form1()
    {
        InitializeComponent();
    }

    private void asapBtn_Click(object sender, EventArgs e)
    {
        cs.setStrategy(new DoneQuick());
        displayBox.Text = cs.getStrategy();
    }

    private void proBtn_Click(object sender, EventArgs e)
    {
        cs.setStrategy(new Procrastinate());
        displayBox.Text = cs.getStrategy();
    }

    private void breaksBtn_Click(object sender, EventArgs e)
    {
        cs.setStrategy(new Breaks());
        displayBox.Text = cs.getStrategy();
    }

    private void blowBtn_Click(object sender, EventArgs e)
    {
        cs.setStrategy(new BlowOff());
        displayBox.Text = cs.getStrategy();
    }
}
```

The coding for the form was by far the easiest part of this program, as I was just piecing the parts together at this point. First I define a new instance of ConcreteStrategy, since every button will be using it in one way or another. Each of the buttons when clicked, sends in the corresponding strategy object, and the text is updated to fit the strategy that is requested.

Conclusion:

In my opinion, this has been the easiest pattern so far. This pattern came to me so easily, and the few online resources I used helped me get a better understanding of the various ways this pattern could be used. It is definitely a pattern that I could see myself using in the future as a prestigious developer.