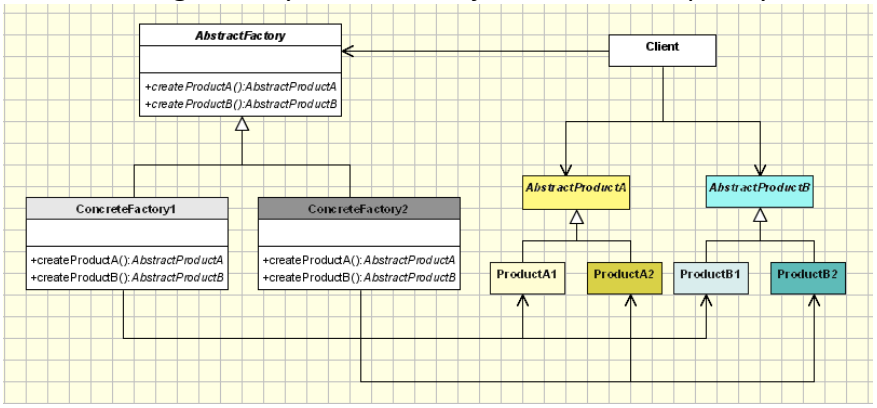
Alex Lang

The Abstract Factory Pattern

Design Patterns

9/27/16

For this assignment, we were told to create a program the utilized the abstract factory pattern. One of the ways this design pattern can be described according to OODesign is as a "super factory". This super factory creates other factories which create objects themselves. This Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes. For my program, I chose to make a clothing factory, which divided into a shirt factory, and a pants factory. The UML diagram for the Abstract Factory Pattern is pictured above. The abstract factory in my case, was a generic clothing factory. The concrete factories I had were shirt factories and pants factories. My abstract product in this case was clothes, and my concrete products were shirts and pants.



I had classes for the following:

Clothes (Abstract Product) ClothesFactory (Abstract Factory)

Pants (Product) PantsFactory(Concrete Factory)

Shirt(Product) ShirtFactory(Concrete Factory)

The first class I started with was the Clothes class. I started here because I needed to start listing all of the qualities all clothing has. This was an abstract class, and here is where I declared some functions that all items of clothing have.

```
public abstract class Clothes //abstract product class for clothes
    {
        public abstract string getBrand();
        public abstract string getColor();
        public abstract string getSize();
        public abstract string getType();
        public abstract string getDescription();
    }
```

Next, I made the shirt class which inherits from the Clothes class. Here is where I added to the inherited functions and really defined what the shirt object consists of.

```
public class Shirt : Clothes //Our shirt product, inherits from clothes
    {

        private string brand;
        private string color;
        private string size;
        private string type;
        private string pattern

        //Constructor for shirt
        public Shirt(string type, string brand, string color, string size, string pattern)
        {
            this.type = type;
            this.brand = brand;
            this.color = color;
            this.size = size;
            this.pattern = pattern;
        }

        public override string getBrand()  //Return values passed into shirt constructor
        {
            return brand;
        }

        public override string getColor()
        {
            return color;
        }

        public override string getSize()
        {
            return size;
        }

        public override string getType()
        {
            return type;
        }

        public override string getDescription()
```

```
        {
            return "You purchased a " + color + " " + type + " made by " + brand + " in size " + size + "
with a " + pattern + " pattern.";
        }

        public string getPattern()
        {
            return pattern;
        }

    }
```

The only function that was added to the shirt class was the getPattern function. A lot of shirts have patterns on them, and most pants usually do not, so I thought it would be fitting to implement this into the shirts class. The constructor serves to define what kind of shirt is being created. The getDescription method is all of the qualities the shirt has, entered into a string I created simulating a user buying the item.

Next, I created the Pants class, which is extremely similar to the Shirts class, with only a few minor changes.

```
public class Pants : Clothes //Pants product, inherits from clothes
    {
        private string type;
        private string brand;
        private string color;
        private string size;
        private string width;

        //Constructor for pants

        public Pants(string type, string brand, string color, string size, string width)
        {
            this.type = type;
            this.brand = brand;
            this.color = color;
            this.size = size;
            this.width = width;
        }
        public override string getBrand() //Return values passed into pants constructor
        {
            return brand;
        }

        public override string getColor()
        {
            return color;
        }

        public override string getSize()
        {
            return size;
        }
```

```
        public override string getType()
        {
            return type;
        }

        public override string getDescription()
        {
            return "You purchased " + color + " " + type + " made by " + brand + " in size " + size + "
that fit " + width + ".";
        }

        public string getWidth()
        {
            return width;
        }

    }
```

The only differences in this class from the shirt class is that this creates a pants object instead of a shirt object, the description is slightly altered, and there is the getWidth() method. Pants come in many different widths, and that is why I chose to implement this method.

Next I created the factories for these products. I started off with the ClothingFactory, which is the abstract factory. I only had a single method in this class since both concrete factories have one major thing in common, they both overall just create clothing.

```
public abstract class ClothesFactory //Abstract Factory, creates other factories
    {
         public abstract Clothes createClothing();
    }
```

With the abstract factory created, I then moved on and create the two concrete factories, ShirtFactory and PantsFactory. The ShirtFactory inherites from the ClothesFactory class.

```
public class ShirtFactory : ClothesFactory
    {
        private string brand;
        private string color;
        private string size;
        private string type;
        private string pattern;

        //Constructor for ShirtFactory
        public ShirtFactory(string type, string brand, string color, string size, string pattern)
        {
```

```
            this.type = type;
            this.brand = brand;
            this.color = color;
            this.size = size;
            this.pattern = pattern;
        }

        public override Clothes createClothing()
        {
            return new Shirt(type, brand, color, size, pattern);
            //Creates new shirt object from values passed in
        }
    }
```

Here is where the actual Shirt object is created with the values passed in. The PantsFactory is similar, but the only difference being width passed in instead of pattern.

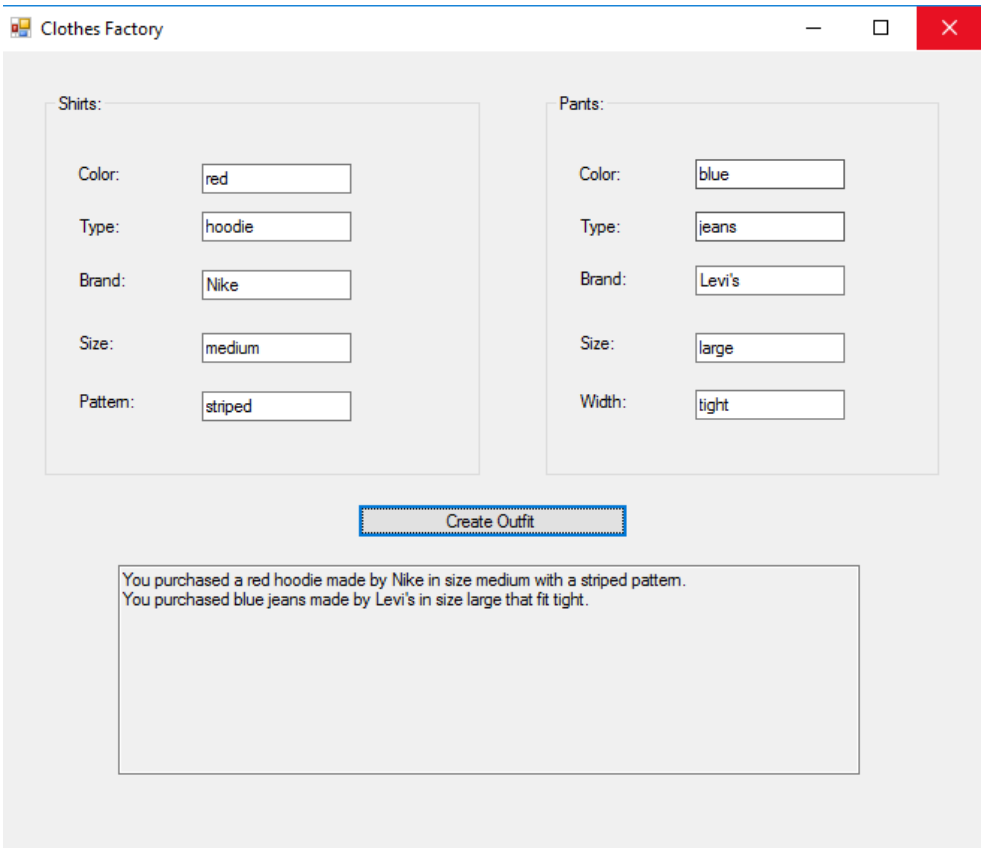
```
public class PantsFactory : ClothesFactory
    {
        private string brand;
        private string color;
        private string size;
        private string type;
        private string width;

        //constructor for PantsFactory
        public PantsFactory(string type, string brand, string color, string size, string
width)
        {
            this.type = type;
            this.brand = brand;
            this.color = color;
            this.size = size;
            this.width = width;
        }

        public override Clothes createClothing()
        {
            return new Pants(type, brand, color, size, width);
            //Creates new Pants object from values passed in
        }
    }
```

Finally, all I had to do was put everything together in the form. This was very easy since the factories pretty much did all the work for me already.

```
public partial class Form1 : Form
    {
        //Creating new instances of these classes
        ShirtFactory SF;
        PantsFactory PF;
        Clothes clothes;

        public Form1()
```

```
        {
            InitializeComponent();
        }

        private void createBtn_Click(object sender, EventArgs e)
        {
            //Send in user input to the shirt factory, make new clothing item out of it, update textbox.
            SF = new ShirtFactory(shrtTypeBox.Text, shrtBrandBox.Text, shrtClrBox.Text, shrtSizeBox.Text,
shrtPatternBox.Text);
            clothes = SF.createClothing();
            productBox.Text += clothes.getDescription() + Environment.NewLine;

            //Send in user input to the pants factory, make new clothing item out of it, update textbox.
            PF = new PantsFactory(pntsTypeBox.Text, pntsBrandBox.Text, pntsClrBox.Text, pntsSizeBox.Text,
pntsWidthBox.Text);
            clothes = PF.createClothing();
            productBox.Text += clothes.getDescription() + Environment.NewLine + Environment.NewLine;
        }
    }
```

Before anything else I create new instances of the shirt factory, the pants factory and the clothes. When the button is pressed, a new shirt factory is created with the user specifications, and the clothes object is created from the factory. The same is done for the pants factory, and all of the clothes created are described on the main form.



Conclusion: This design pattern was pretty enjoyable to write, and for some reason I had a much better understanding of this pattern than the factory method pattern. There were a lot more resources that explained this pattern, and the way OODesign described it, as a “super factory” made creating this programming so much easier.