Design Patterns

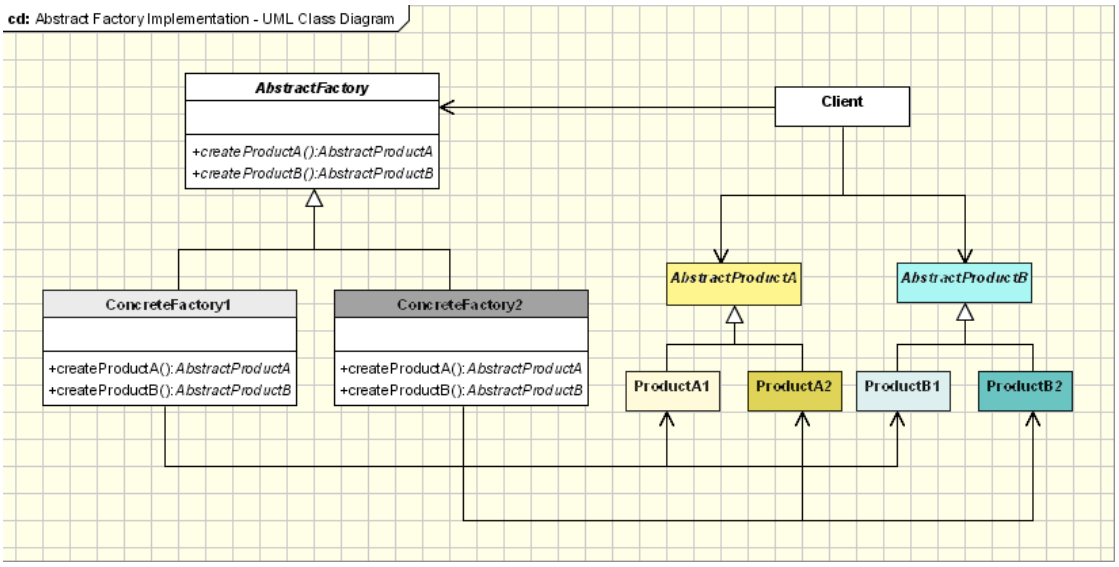
Abstract Factory Pattern

Collin Kemner

September 29th, 2016

Abstract Factory Pattern

Introduction



The purpose of this assignment is to showcase the Abstract Factory pattern. The Abstract Factory pattern is different from the Factory Method pattern as this pattern uses abstract factory and product classes to create its product. Since they're both "factory patterns", they're easy to get mixed up with.

UML Diagram

The UML diagram showcases how the Abstract Factory pattern works. As described by OODesign, the Abstract Factory pattern "offers the interface for creating a family of related objects without explicitly specifying their classes." So, the abstract factory class declares the interface for operations in regards to abstract products while the concrete factory implements operations used to create the concrete products. Next, the abstract product creates the interface for the products and the product class defines the product that is to be made by corresponding with the concrete factory class and abstract product class.

The Application and Code

The application that was created to show this pattern mimics a two guitar factories, both that produce a different kind and style of guitar. Below are the classes that are used for this application:

Classes	Description
Abstract Factory (Interface)	There are two factory interfaces which are for the solid body factory and hollow body factory, which creates a method specific to its style of guitar.
Concrete Factory	There are two concrete factories which return its specific guitar depending on which method is referred to.
Abstract Product	There are two abstract products for the two guitar types and they define strings which will describe the different aspects of the guitar.
Product	The guitars (products) that will be described and created throughout the duration of the pattern.

```
public interface AbstractFactorySolid
//factory interface
    {
        SolidBody createSolidBody1();
        SolidBody createSolidBody2();
        SolidBody createSolidBody3();
    }

    public interface
AbstractFactoryHollow
    {
        HollowBody createHollowBody1();
        HollowBody createHollowBody2();
    }
```

First off in the code, two public interfaces are created. They contain methods which refer to the two types of guitars, SolidBody (solid body guitars are guitars that are made of solid wood that do not rely on a normal sound box, but on an electric pickup system) and HollowBody (hollow body guitars are either normal acoustic guitars or electric guitars with a pickup system plus a sound box). As they have different characteristics, both have their own interfaces to differentiate the two a little better.

```
public class ConcreteFactorySolid : AbstractFactorySolid
    //concrete factory solid body guitars
    {
        public SolidBody createSolidBody1()
        {
            return new FenderMustang();
        }
        public SolidBody createSolidBody2()
        {
            return new FenderStratocaster();
        }
        public SolidBody createSolidBody3()
        {
            return new GibsonLesPaul();
        }
    }

    public class ConcreteFactoryHollow : AbstractFactoryHollow
    //concrete factory hollow body guitars
    {
        public HollowBody createHollowBody1()
        {
            return new GibsonES335();
        }
        public HollowBody createHollowBody2()
        {
            return new IbanezAF55();
        }
    }
```

To coincide with the two interfaces, two concrete factories are created for their respective guitar type. For each guitar, the abstract product is referred to in the method which returns the guitars in order of the "create guitar" method. The information from the guitar comes from its guitar (product) class.

```
public abstract class SolidBody
//Abstract product class (SolidBody)
    {
        private string Name;
        private string Weight;
        private string NeckWidth;
        private string BridgeType;
        private string BodyType;
    }

public abstract class HollowBody
//Abstract product interface (HollowBody)
    {
        private string Name;
        private string Weight;
        private string NeckWidth;
        private string BridgeType;
        private string BodyType;
    }
```

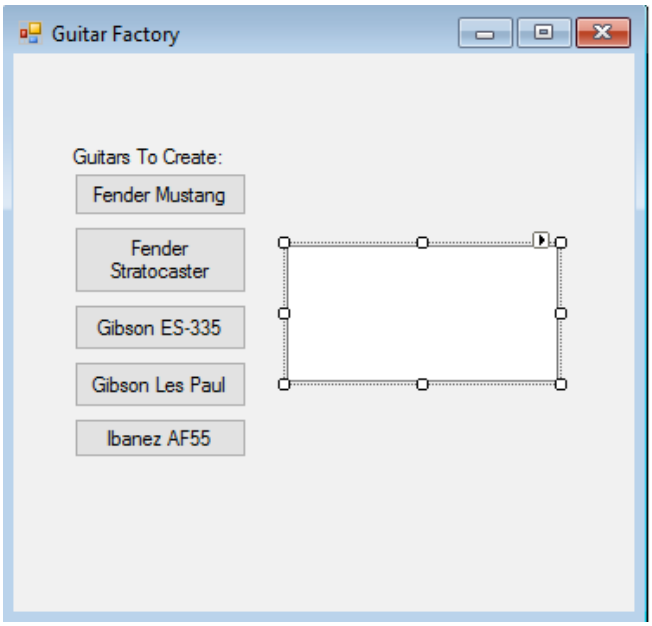
Next are the abstract classes for both the solid body and hollow body guitars. The classes initialize the strings of the descriptions of the guitars. These two classes are referred to by the factory interfaces and the concrete factory classes as a part of putting the descriptions of the guitars together.

The last classes are the guitar (product) classes. They contain the complete description of the guitars in the respective strings and are put altogether in a ToString method which is overridden. The full description is returned and is able to be sent to the application from the ToString method:

```
public class FenderMustang : SolidBody
//fender mustang class (product class)
    {
        public string Name = "Fender Mustang";
        public string Weight = "~6 lbs";
        public string NeckWidth = "Slim";
        public string BridgeType = "Tremolo";
        public string BodyType = "Solid";

        public override string ToString()
        {
            string FullDescription = "Guitar Produced:" + "\r\n" + "Name: " + Name +
"\r\n" + "Weight: " + Weight + "\r\n" + "Neck Style: " + NeckWidth + "\r\n" + "Bridge
Type: " + BridgeType + "\r\n" + "Body Type: " + BodyType;

            return FullDescription;
        }
    }
    public class FenderStratocaster : SolidBody
//fender stratocaster class (product class)
    {
        public string Name = "Fender Stratocaster";
        public string Weight = "~6-7 lbs";
        public string NeckWidth = "Slim";
        public string BridgeType = "Fixed or Tremolo";
        public string BodyType = "Solid";

        public override string ToString()
        {
            string FullDescription = "Guitar Produced:" + "\r\n" + "Name: " + Name +
"\r\n" + "Weight: " + Weight + "\r\n" + "Neck Style: " + NeckWidth + "\r\n" + "Bridge
Type: " + BridgeType + "\r\n" + "Body Type: " + BodyType + "\r\n";

            return FullDescription;
        }
    }
    public class GibsonLesPaul : SolidBody
//gibson les paul class (product class)
    {
        public string Name = "Gibson Les Paul";
        public string Weight = "~ 11 lbs";
        public string NeckWidth = "Fat";
        public string BridgeType = "Fixed";
        public string BodyType = "Solid";

        public override string ToString()
        {
            string FullDescription = "Guitar Produced:" + "\r\n" + "Name: " + Name +
"\r\n" + "Weight: " + Weight + "\r\n" + "Neck Style: " + NeckWidth + "\r\n" + "Bridge
Type: " + BridgeType + "\r\n" + "Body Type: " + BodyType + "\r\n";
```

```
            return FullDescription;
        }
    }

public class GibsonES335 : HollowBody   //gibson es-335 class (product class)
    {
        public string Name = "Gibson ES-335";
        public string Weight = "~7 lbs";
        public string NeckWidth = "Fat";
        public string BridgeType = "Fixed or Floating";
        public string BodyType = "Hollow";

        public override string ToString()
        {
            string FullDescription = "Guitar Produced:" + "\r\n" + "Name: " + Name +
"\r\n" + "Weight: " + Weight + "\r\n" + "Neck Style: " + NeckWidth + "\r\n" + "Bridge
Type: " + BridgeType + "\r\n" + "Body Type: " + BodyType + "\r\n";

            return FullDescription;
        }
    }
    public class IbanezAF55 : HollowBody   //ibanez af55 class (product class)
    {
        public string Name = "Ibanez AF55";
        public string Weight = "~5-6 lbs";
        public string NeckWidth = "Fat";
        public string BridgeType = "Floating";
        public string BodyType = "Hollow";

        public override string ToString()
        {
            string FullDescription = "Guitar Produced:" + "\r\n" + "Name: " + Name +
"\r\n" + "Weight: " + Weight + "\r\n" + "Neck Style: " + NeckWidth + "\r\n" + "Bridge
Type: " + BridgeType + "\r\n" + "Body Type: " + BodyType + "\r\n";

            return FullDescription;
        }
```

```
    }
```



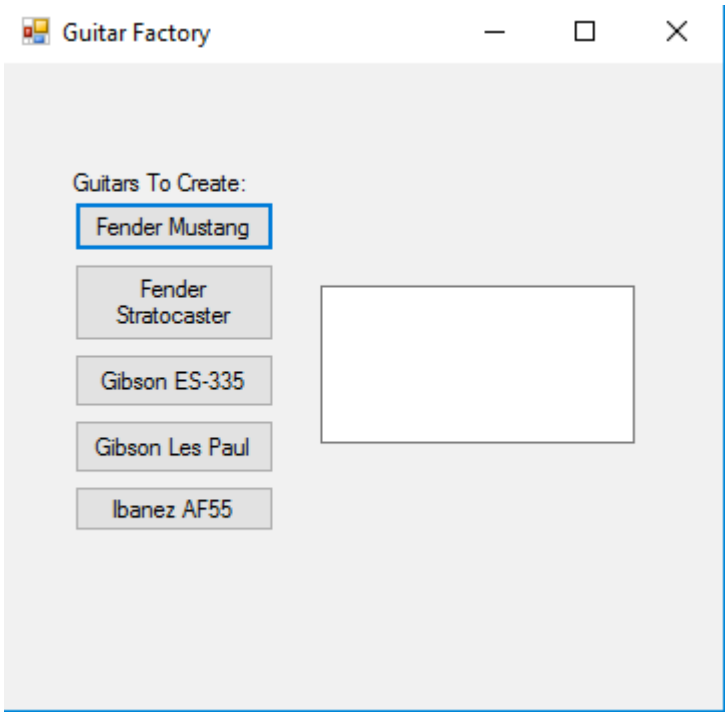
The photo on the side shows what the application with GUI looks like. It has five buttons for five different guitars. There is also a text box which is where the information for the guitars are pasted. A list box was at first being used, but a text box works better with wrapping text and is easier to work with.

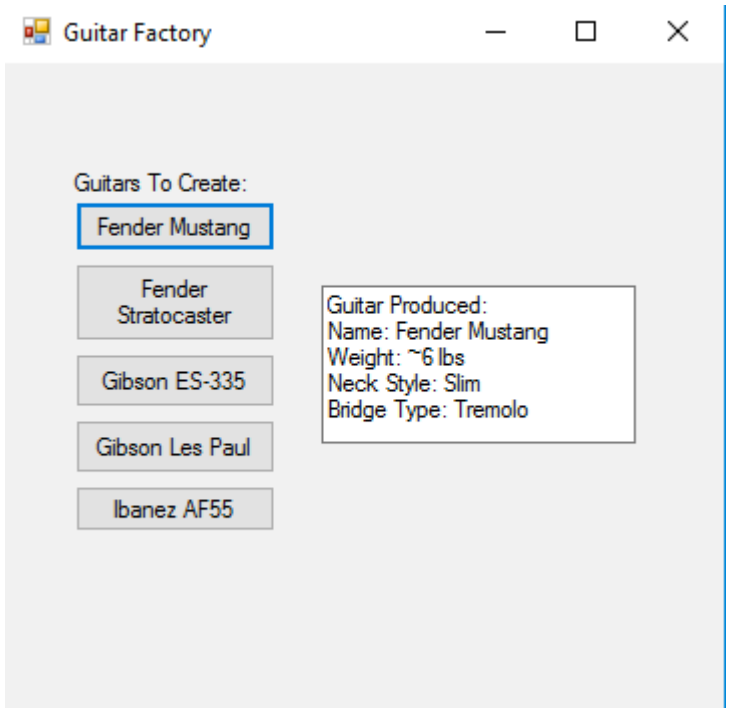
The form code is as appears below. It's pretty basic as it's only button click events for the five guitars. In each click event, the ToString method is referred to and stored as a string which is then set to the text box as text:

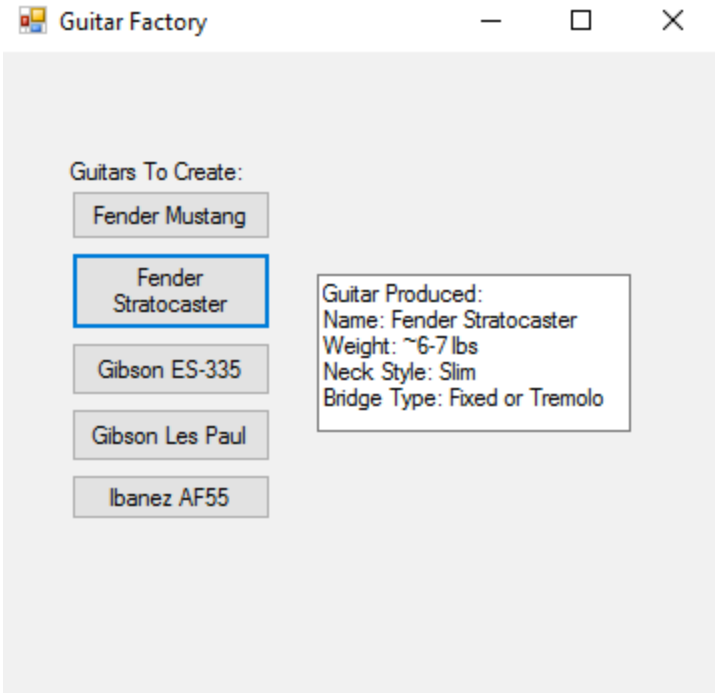
```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void m_btnMustang_Click(object sender, EventArgs e)
        {
            AbstractFactorySolid guitarFactory = new ConcreteFactorySolid();
            string factoryString = guitarFactory.createSolidBody1().ToString();

            m_tbFactoryBox.Text = factoryString;
        }

        private void m_btnStratocaster_Click(object sender, EventArgs e)
        {
            AbstractFactorySolid guitarFactory = new ConcreteFactorySolid();
            string factoryString = guitarFactory.createSolidBody2().ToString();

            m_tbFactoryBox.Text = factoryString;
        }

        private void m_btnES335_Click(object sender, EventArgs e)
        {
            AbstractFactoryHollow guitarFactory = new ConcreteFactoryHollow();
            string factoryString = guitarFactory.createHollowBody1().ToString();

            m_tbFactoryBox.Text = factoryString;
        }

        private void m_btnLesPaul_Click(object sender, EventArgs e)
```

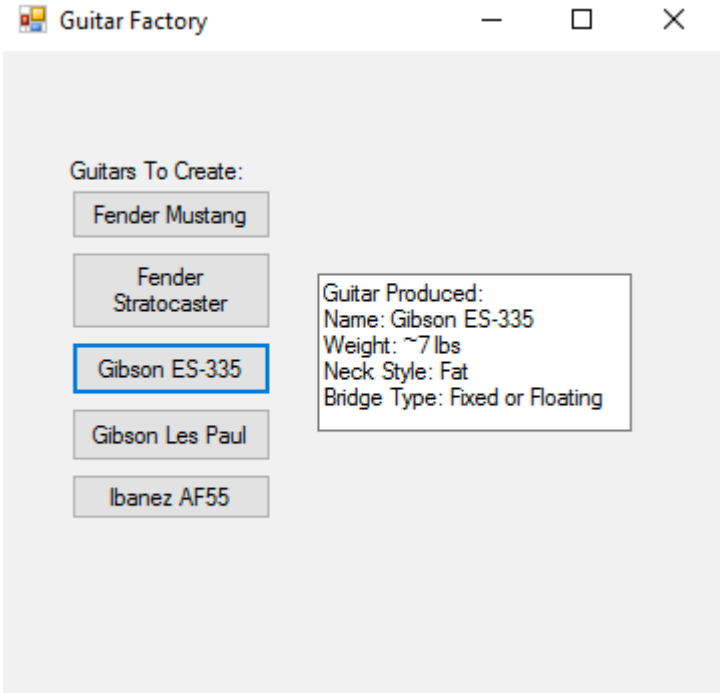
```
        {
            AbstractFactorySolid guitarFactory = new ConcreteFactorySolid();
            string factoryString = guitarFactory.createSolidBody3().ToString();

            m_tbFactoryBox.Text = factoryString;
        }

        private void m_btnAF55_Click(object sender, EventArgs e)
        {
            AbstractFactoryHollow guitarFactory = new ConcreteFactoryHollow();
            string factoryString = guitarFactory.createHollowBody2().ToString();

            m_tbFactoryBox.Text = factoryString;
        }
    }
```

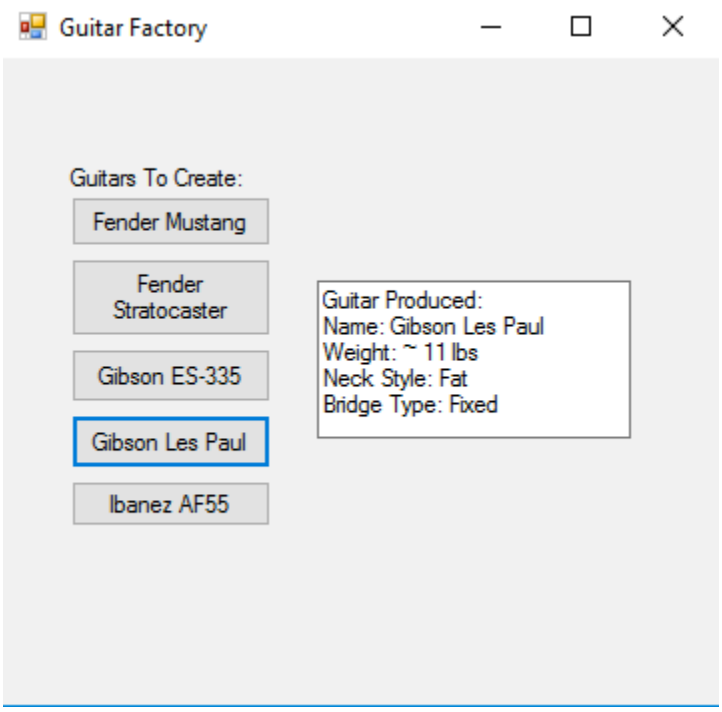
As proof that the application works, the following screenshots display the program working and giving the descriptions of a specific guitar when a specific button is pressed:

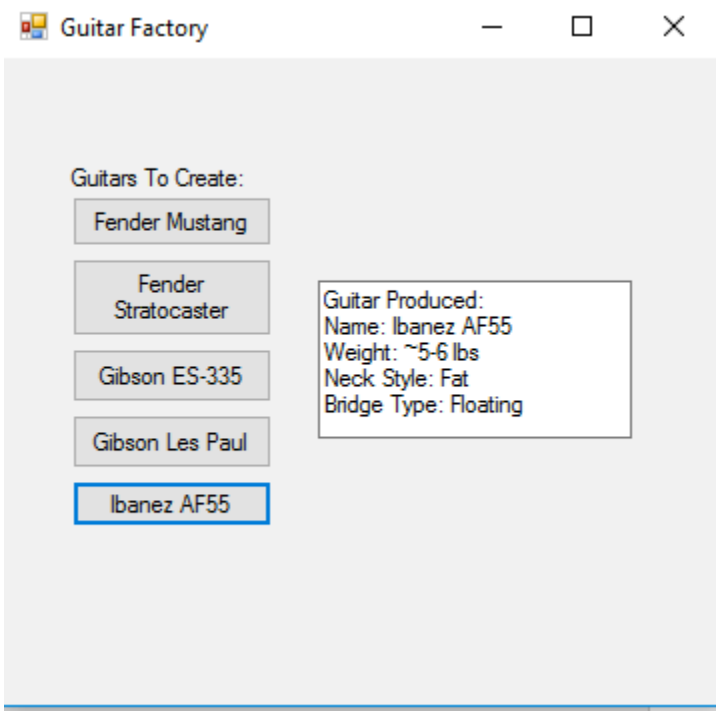

Guitar Factory
Guitars To Create:
Fender Mustang
Fender Stratocaster
Gibson ES-335
Gibson Les Paul
Ibanez AF55


Guitar Factory
Guitars To Create:
Fender Mustang
Fender Stratocaster
Gibson ES-335
Gibson Les Paul
Ibanez AF55
Guitar Produced:
Name: Fender Mustang
Weight: ~6 lbs
Neck Style: Slim
Bridge Type: Tremolo


Guitar Factory
Guitars To Create:
Fender Mustang
Fender Stratocaster
Gibson ES-335
Gibson Les Paul
Ibanez AF55
Guitar Produced:
Name: Fender Stratocaster
Weight: ~6-7 lbs
Neck Style: Slim
Bridge Type: Fixed or Tremolo


Guitar Factory
Guitars To Create:
Fender Mustang
Fender Stratocaster
Gibson ES-335
Gibson Les Paul
Ibanez AF55
Guitar Produced:
Name: Gibson ES-335
Weight: ~7 lbs
Neck Style: Fat
Bridge Type: Fixed or Floating





As seen in the above screenshots, when a button is pressed, it shows the description of the guitar and that guitar only. It does not show multiple productions at the same time, only that one that has been pressed.

Observations and Reflections

In the end, I ended up enjoying this pattern. I am aware that I submitted it a little late, but that's because I was having some issues with certain aspects of the pattern. However, I ended up making it work on my own and without the help of students or the professor, so that I'm pretty happy with. Normally I need help from an outside source and this time I generally did it on my own. As a result of that, I believe that I learned more than doing any other pattern so far. I hope that what I learned will help carry me forward in the other patterns that I do. I'm not the greatest with doing code like this completely on my own so I see this as a milestone in my understanding of creating patterns and the C# language.