Jackson Spencer

The State Pattern

November 30, 2016

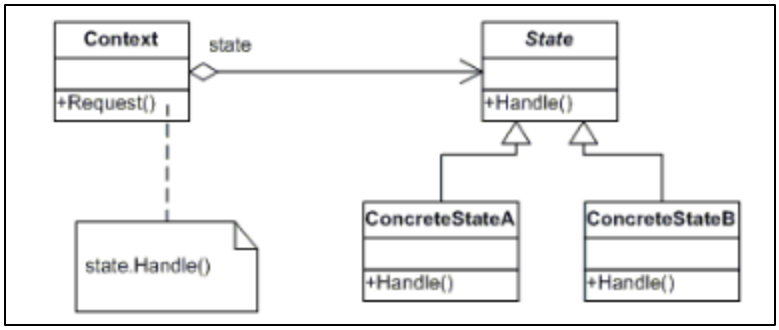
ECCS 2411: Design Patterns

Introduction

The purpose of this pattern is to allow an object to alter its behavior based on the state that it is in. This is demonstrated in my application in the form of an electric guitar amplifier. The user selects what type of sound they wish to produce, then the levels are adjusted accordingly to fit the desired sound.

UML Diagram

The UML diagram displays the necessary components and functions for the pattern. The Context class contains a reference to the State interface. Any ConcreteState class inherits from the State interface as well. The Context class in my application is called Amplifier, the State class is called Sound, and the Concrete states are Blues, Bright, and Metal.



Classes

The Amplifier class contains a reference to a Sound object, and defines four variables for the sound. Drive, Bass, Middle, and Treble. When the object changes state, the string is updated to display the new results.

```
public class Amplifier      // The Context Class
    {
        // The Amplifier starts in a very neutral setting
        private int Drive = 5;
        private int Bass = 3;
        private int Middle = 5;
        private int Treble = 5;

        private Sound _sound = new Blues();

        public void neutral()
        {
            int[] changes = _sound.makeSoundNeutral();
            handleChanges(changes);
            changeSound();
        }

        public void mellow()
        {
            int[] changes = _sound.makeSoundMellow();
            handleChanges(changes);
            changeSound();
        }

        public void happy()
        {
            int[] changes = _sound.makeSoundHappy();
            handleChanges(changes);
            changeSound();
        }

        public void heavy()
        {
            int[] changes = _sound.makeSoundHeavy();
            handleChanges(changes);
            changeSound();
        }

        public override string ToString()
        {
            return "Your settings will produce a sound of " + _sound.ToString() +
".\n\nDrive level is "
                + Drive + "/11.\n\nBass level is " + Bass + "/11.\n\nMid level is " +
Middle +
                "/11.\n\nTreble level is " + Treble + "/11." ;
        }

        private void changeSound()
        {
            if (Drive >= 8)
                _sound = new Metal();
            else if (Treble <= 3)
                _sound = new Blues();
            else if (Bass <= 4)
                _sound = new Bright();
        }

        private void handleChanges(int[] changes)
```

```
        {
            Drive += changes[0];
            Bass += changes[1];
            Middle += changes[2];
            Treble += changes[3];

            if (Drive > 10)
                Drive = 10;
            if (Drive < 0)
                Drive = 0;
            if (Bass > 10)
                Bass = 10;
            if (Bass < 0)
                Bass = 0;
            if (Middle > 10)
                Middle = 10;
            if (Middle < 0)
                Middle = 0;
            if (Treble > 10)
                Treble = 10;
            if (Treble < 0)
                Treble = 0;
        }
    }
```

The Sound class is abstract, in place of being an interface. It creates prototypes to handle the four types of sound: Neutral, Mellow, Happy, and Heavy.

```
public abstract class Sound     // The State Interface
    {
        public abstract int[] makeSoundNeutral();
        public abstract int[] makeSoundMellow();
        public abstract int[] makeSoundHappy();
        public abstract int[] makeSoundHeavy();
    }
```

The Concrete State classes are all very similar and all three are shown here. Each concrete state changes based on the current sound being produced. For example, when the current sound is bright, more bass will need to be added to produce a blues sound.

Blues

```
public class Blues: Sound       // A concrete state class
    {
        public override int[] makeSoundNeutral()
        {
            return new int[4] { 1, -3, 1, 2 };
        }
```

```
        public override int[] makeSoundMellow()
        {
            return new int[4] { 1, 2, 1, -2 };
        }

        public override int[] makeSoundHappy()
        {
            return new int[4] { -2, -3, 1, 3 };
        }

        public override int[] makeSoundHeavy()
        {
            return new int[4] { 2, 1, -3, 1 };
        }

        public override string ToString()
        {
            return " the Blues";
        }
    }
```

Bright

```
public class Bright: Sound      // A concrete state class
    {
        public override int[] makeSoundNeutral()
        {
            return new int[4] { 1, 2, -1, -2 };
        }

        public override int[] makeSoundMellow()
        {
            return new int[4] { 1, 2, -1, -1 };
        }

        public override int[] makeSoundHappy()
        {
            return new int[4] { -1, -2, 1, 2 };
        }

        public override int[] makeSoundHeavy()
        {
            return new int[4] { 3, 3, -2, 1 };
        }

        public override string ToString()
        {
            return " clean and natural brightness";
        }
    }
```

Metal

```
public class Metal: Sound       // A concrete state class
    {
        public override int[] makeSoundNeutral()
        {
            return new int[4] { -3, -3, 3, -2 };
        }

        public override int[] makeSoundMellow()
        {
            return new int[4] { -1, 1, 1, -1 };
        }

        public override int[] makeSoundHappy()
        {
            return new int[4] { -3, -5, 2, 2 };
        }

        public override int[] makeSoundHeavy()
        {
            return new int[4] { 2, 2, -3, 2 };
        }

        public override string ToString()
        {
            return " a METAL variety";
        }
    }
```

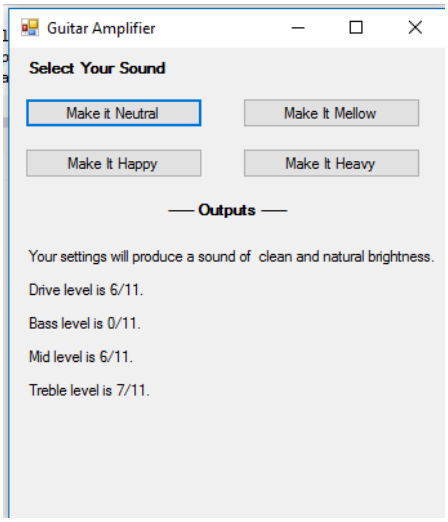
The only thing the form is responsible for, is giving the classes functionality. Each button updates the label based on its associated method. I created an Amplifier object “amp” to be used in each button click event.

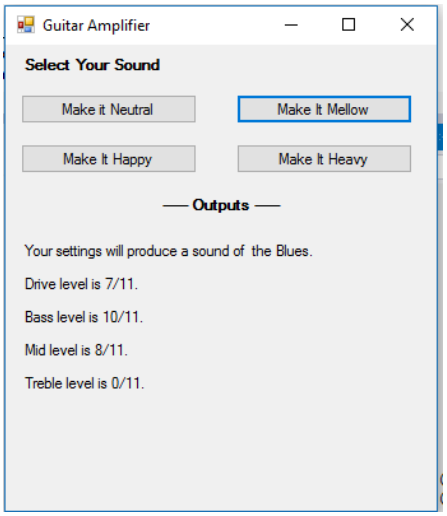
```
public partial class Form1 : Form
    {
        Amplifier amp = new Amplifier();

        public Form1()
        {
            InitializeComponent();
        }

        private void btn_Neutral_Click(object sender, EventArgs e)
        {
            amp.neutral();
            lbl_Output.Text = amp.ToString();
        }

        private void btn_Mellow_Click(object sender, EventArgs e)
        {
            amp.mellow();
```

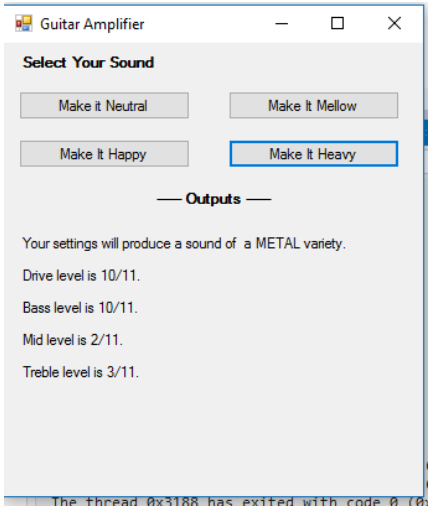
```
            lbl_Output.Text = amp.ToString();
        }

        private void btn_Happy_Click(object sender, EventArgs e)
        {
            amp.happy();
            lbl_Output.Text = amp.ToString();
        }

        private void btn_Heavy_Click(object sender, EventArgs e)
        {
            amp.heavy();
            lbl_Output.Text = amp.ToString();
        }
    }
```

Captures

These images display the application in action after changing state using the associated buttons.







Reflection

First, I give credit to Andreya for helping me develop an idea for the application. Her different levels of emotions gave me the idea of level settings on an amplifier. This pattern was difficult when it came to assigning values to the methods in the Concrete State classes. I still believe that the way I did it was not the simplest, and I am looking into an alternative. I see the value of this pattern, and I believe that I will be able to recognize it in the future.