Design Patterns

Bridge Pattern

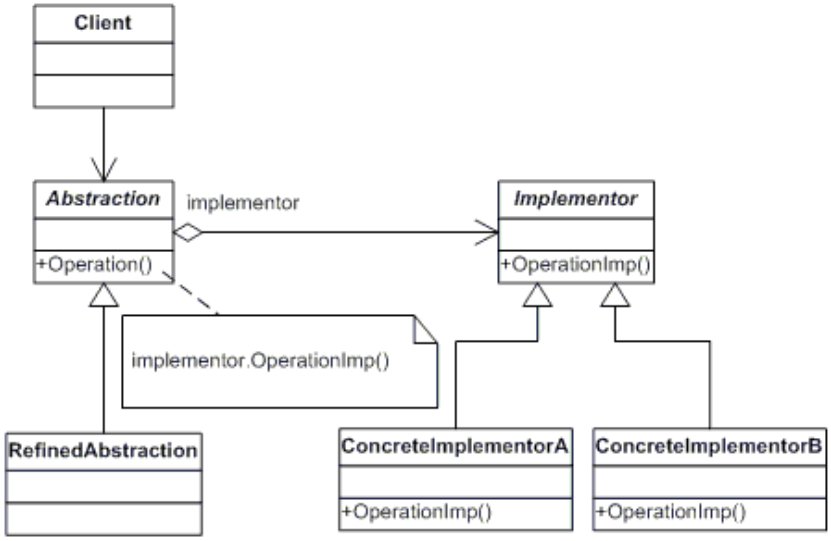
Collin Kemner

November 9th, 2016

Introduction

This is assignment will demonstrate and show the Bridge pattern. The application designed for this assignment is a music player simulation. The app will not play music, but will show the name of the album being played and its genre. In a sense, it's a skeleton of a music player without the guts of it included in the application.

UML Diagram



According to OODesign.com, the Bridge Pattern "decouple(s) abstraction from implementation so that the two can vary independently". The Abstraction class defines an abstract interface with an Implementor object while the RefinedAbstraction class implements the Abstraction interface. The Implementor interface defines an interface for the Implementor classes and may be different than the Abstraction interface, although the two work towards a common goal. Finally, the ConcreteImplementor's implement the Implementor interface.

Application code: Implementor Interface

```
public interface Implementor   //Implementor class
    {
        string GetMusicGenre();
        string GetMusicAlbum();
    }
```

Starting off in the code is the Implementor interface. The interface defines a couple of string methods. The "GetMusicGenre" method will hold the music genre chosen and the "GetMusicAlbum" method will hold the album name that is randomly selected.

Concrete Implementor Classes

```
#region GarageRock class

    public class GarageRock : Implementor   //Concrete Implementor class
    {
        static Random rand = new Random();

        const string INTRO = "Genre: ";
        const string PLAY = "Now playing: ";

        public List<string> _musicAlbum = new List<string>();

        public GarageRock()
        {
            _musicAlbum.Add("Superfuzz Bigmuff by Mudhoney");
            _musicAlbum.Add("Twins by Ty Segall");
            _musicAlbum.Add("Here Be Thee Gories by The Gories");
            _musicAlbum.Add("The Traditional Fools by The Traditional Fools");
            _musicAlbum.Add("Help by Thee Oh Sees");
            _musicAlbum.Add("Still Life of Citrus and Slime by CFM");
            _musicAlbum.Add("Moonhearts by Charlie and the Moonhearts");
            _musicAlbum.Add("Bass Drum of Death by Bass Drum of Death");
        }

        public string randomString()
        {
            int Index = rand.Next(_musicAlbum.Count);
            string randomString = _musicAlbum[Index];
            return randomString;
        }

        string Implementor.GetMusicGenre()
        {
            return INTRO + "Garage Rock";
        }
        string Implementor.GetMusicAlbum()
        {
            return PLAY + randomString();
        }
    }

    #endregion
```

Next are the Concrete Implementor classes. There are three of them and they are all different genres of rock: Garage, Classic, and Punk. In the case of the Garage Rock genre class, a list of eight albums are defined. Then, a Random class object is defined so that a random element of the list is chosen (this happens in the “randomString” string method). When the random album is selected, the GetMusicGenre string method from the Implementor class is called and set to return the constant string “INTRO” as well as its genre. Next, the second string method from the Implementor class is called and returns the constant string “PLAY” as well as “randomString”, the randomly selected album. This exact sequence of events happens for the two other Concrete Implementor classes.

```
#region ClassicRock class

public class ClassicRock : Implementor //Concrete Implementor class
{
    static Random rand = new Random();

    const string INTRO = "Genre: ";
    const string PLAY = "Now playing: ";

    public List<string> _musicAlbum = new List<string>();

    public ClassicRock()
    {
        _musicAlbum.Add("Led Zeppelin II by Led Zeppelin");
        _musicAlbum.Add("Pipers At The Gates Of Dawn by Pink Floyd");
        _musicAlbum.Add("Electric Ladyland by Jimi Hendrix");
        _musicAlbum.Add("Paranoid by Black Sabbath");
        _musicAlbum.Add("Cheap Thrills by Big Brother & The Holding Company");
        _musicAlbum.Add("Sgt. Pepper's Lonely Hearts Club Band by The Beatles");
        _musicAlbum.Add("Waiting For The Sun by The Doors");
        _musicAlbum.Add("Part One by The West Coast Pop Art Experimental Band");
    }

    public string randomString()
    {
        int Index = rand.Next(_musicAlbum.Count);
        string randomString = _musicAlbum[Index];
        return randomString;
    }

    string Implementor.GetMusicGenre()
    {
        return INTRO + "Classic Rock";
    }
    string Implementor.GetMusicAlbum()
    {
        return PLAY + randomString();
    }
}

#endregion

#region PunkRock class

public class PunkRock : Implementor //Concrete Implementor class
{
    static Random rand = new Random();

    const string INTRO = "Genre: ";
    const string PLAY = "Now playing: ";

    public List<string> _musicAlbum = new List<string>();

    public PunkRock()
    {
        _musicAlbum.Add("GØGGS by GØGGS");
        _musicAlbum.Add("Dirty by Sonic Youth");
        _musicAlbum.Add("Bleach by Nirvana");
```

```
            _musicAlbum.Add("Nevermind the Bollocks, Here's the Sex Pistols by Sex
        Pistols");
            _musicAlbum.Add("Milo Goes To College by Descendents");
            _musicAlbum.Add("Rocket To Russia by The Ramones");
            _musicAlbum.Add("Damaged by Black Flag");
            _musicAlbum.Add("Living in Darkness by Agent Orange");
        }

        public string randomString()
        {
            int Index = rand.Next(_musicAlbum.Count);
            string randomString = _musicAlbum[Index];
            return randomString;
        }

        string Implementor.GetMusicGenre()
        {
            return INTRO + "Punk Rock";
        }
        string Implementor.GetMusicAlbum()
        {
            return PLAY + randomString();
        }
    }

    #endregion
```

Abstraction Class

```
public abstract class Abstraction  //Abstraction class
    {
        protected Implementor imp;

        public Implementor implementor
        {
            get { return imp; }
            set { imp = value; }
        }

        public virtual string getAllInfo()
        {
            string info = imp.GetMusicGenre() + System.Environment.NewLine +
imp.GetMusicAlbum();
            return info;
        }
    }
```

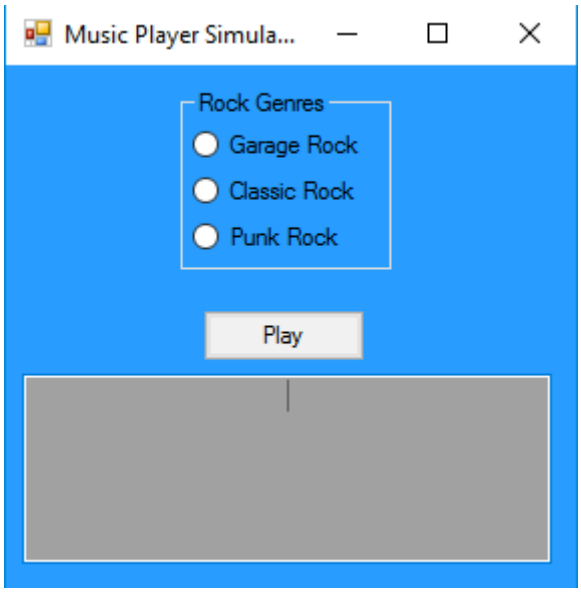
The Abstraction class is an abstract class which contains an Implementor object and an Implementor constructor. This allows the class to retrieve the Implementor's methods. In order to use them, a public virtual string method is created and the string methods from the Implementor is saved to one single string which is then returned from the "getAllInfo" method.

Refined Abstraction Class

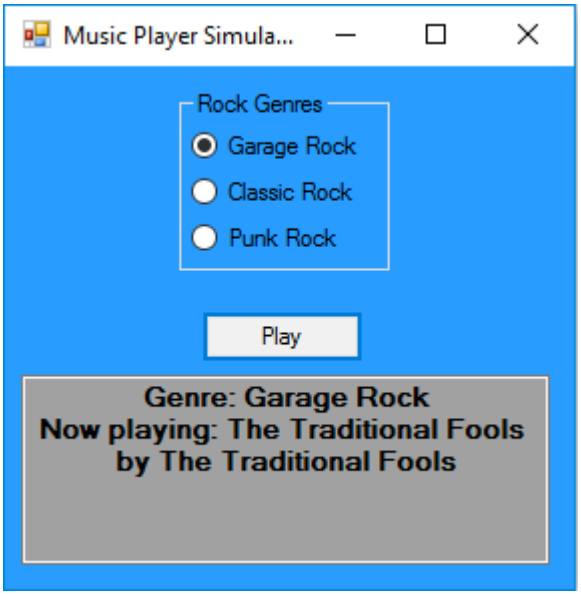
```
public class RefinedAbstraction : Abstraction  //Refined Abstraction (AbstractionImp)
class
    {
        public override string getAllInfo()
        {
            string info = imp.GetMusicGenre() + System.Environment.NewLine +
imp.GetMusicAlbum();
            return info;
        }
    }
```

The Refined Abstraction class implements the Abstraction class and works similarly to the Abstraction although all it contains is the "getAllInfo" method. This time, however, it is an overridden string rather than a virtual string.

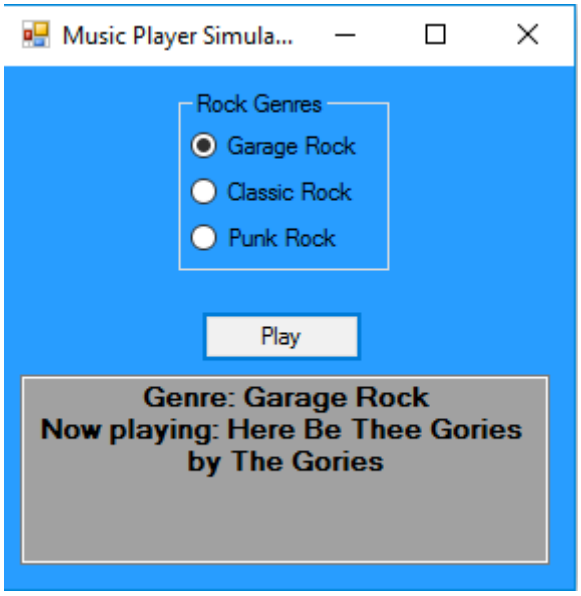
Screenshots As Proof Of Working Application



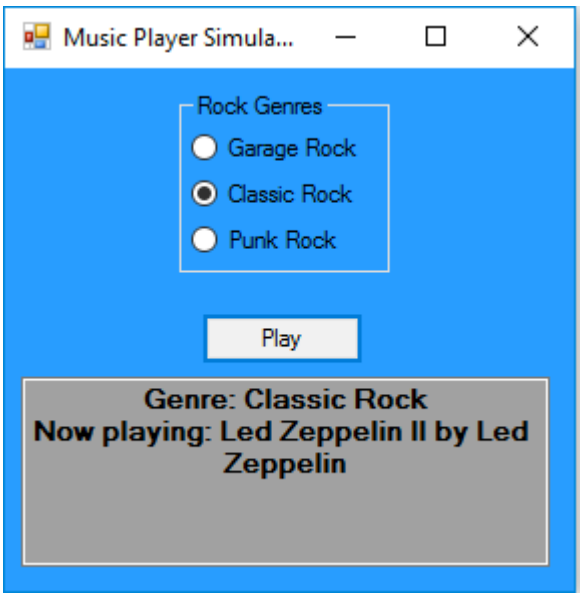
The application when first opened up.



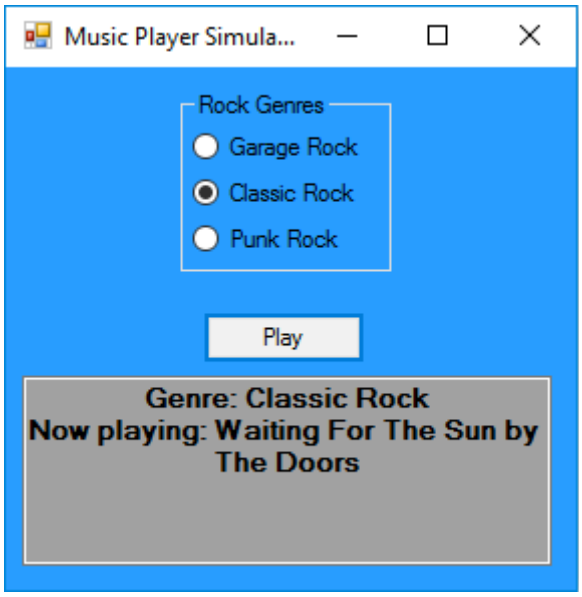
When the "Garage Rock" radio button is selected and the "Play" button is pressed.



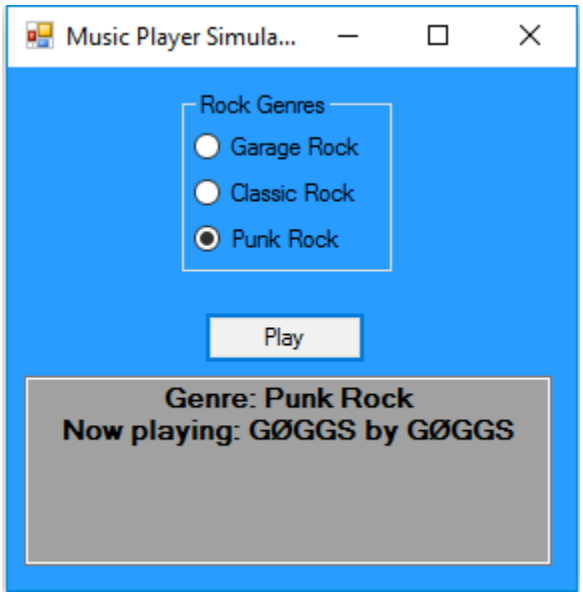
When the "Play" button is pressed again.



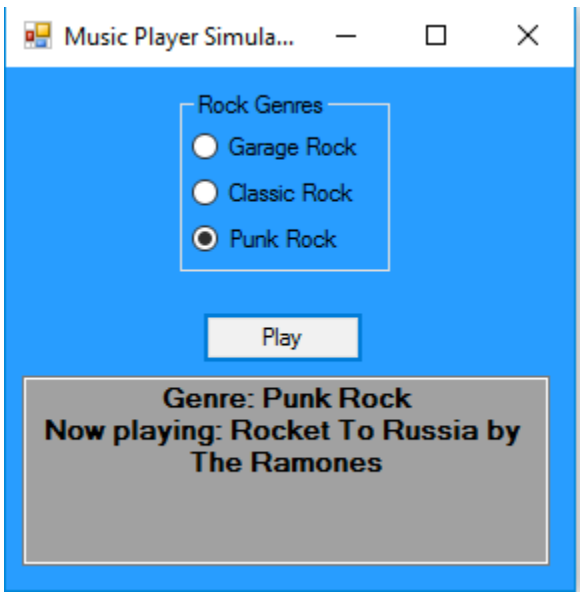
When the "Classic Rock" radio button is selected and the "Play" button is pressed.



When the "Play" button is pressed again.



When the "Punk Rock" radio button is selected and the "Play" button is pressed.



When the "Play" button is pressed again.

Observations and Reflections

This pattern was a fun pattern to write once I figured out what I needed to do in the Abstraction and RefinedAbstraction classes. When I first started writing the code, I started with the Implementor and ConcreteImplementor classes and once I got to the Abstractors, I got confused as to what they were supposed to do and what should be in them. So, ultimately, I left the code be until I found out the answer and finally completed the pattern. Although it is a pretty simple application, I believe that it showcases the Bridge Pattern well and that I will take this information with me to the future. I am feeling that the more I do these patterns, the more I start to understand the structure of C# and coding in general. By this point, I'm starting to feel more confident in my coding (although I still have to look for help on Google now and then). Other than that, things seem to be improving!