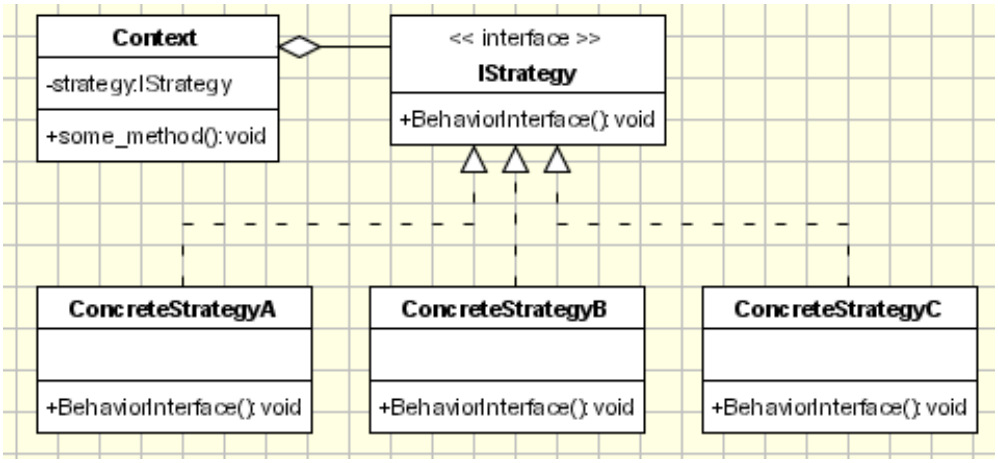
The Strategy Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016-2017

Introduction

This assignment requires an application that makes use of the strategy pattern. This pattern allows the same task to be implemented in several different ways depending on the circumstances. My app displays different types of scales: major, natural minor, harmonic minor, and melodic minor. The user selects the key signature of the scale and what type of scale he would like to see and clicks a display button. A display listbox is updated accordingly.

The UML Diagram



To the left is the UML diagram for the strategy pattern, courtesy of oodesign.com. The Context class calls a method in one of the concrete strategy classes via the IStrategy interface. The method is implemented differently in each of the concrete classes. Below is a table displaying the classes that I used to implement this pattern.

Context	My context class is called Pianist. The pianist plays scales by referring to an object of one of the ConcreteStrategy classes, which is held in an IStrategy variable.
IStrategy	I used an abstract class rather than an interface for my IStrategy class. My IStrategy class is called Scale. It refers to generic scales without a particular type.
ConcreteStrategyA	My first concrete strategy is called MajorScale. It, along with the other concrete strategy classes, implements the methods from the Scale class. The concrete strategy classes are all very similar to one another.
ConcreteStrategyB	My second concrete strategy is called NaturalMinorScale.
ConcreteStrategyC	My third is called HarmonicMinorScale.
ConcreteStrategyD	My final concrete strategy is called MelodicMinorScale.

Narrative and Code

I started writing my code by creating and implementing the abstract scale class. I first provided a prototype for a method called getScaleIndices and described it in comments. This method will be implemented later in each of the concrete strategy classes.

```
public abstract class Scale    // This is the IStrategy interface, implemented here as an
       abstract class
{
    // This method will return the zero-based indices of the appropriate type of scale;
    // For example, if I wanted C - D - E - F - G, I would return 0, 2, 4, 5, 7.
    // I would return the same thing for D - E - F# - G - A.
    // This is based on the pattern of half-steps and whole-steps in the definition of
       each scale type.
    public abstract int[] getScaleIndices();
```

Next, I created and implemented the getScale method. This method is implemented the same way for each concrete strategy class, which is why I chose to use an abstract class instead of an interface for the Scale class. The getScale method returns the notes from the scale as elements of a string array. The scale is returned from bottom to top to bottom. (This is necessary because of the melodic minor scale, which is played differently ascending and descending.) This method is fairly long, but is well-explained throughout by comments.

```
public string[] getScale(int startIndex)
{
    string[] scale = new string[15];
    // Have to use two different arrays - Scales usually use either all sharp or all flat
       notes
    string[] flatNotes = { "A", "B\u266D", "B", "C", "D\u266D", "D", "E\u266D", "E", "F",
       "G\u266D", "G", "A\u266D" };
    string[] sharpNotes = { "A", "A#", "B", "C", "C#", "D", "D#", "E", "F", "F#", "G",
       "G#" };

    // Assume we'll be using flat notes, then change it to sharp notes if we need to
    string[] notes = flatNotes;
    switch (startIndex)
    {
        case 0: case 2: case 4: case 5: case 7: case 9: case 10:
            notes = sharpNotes;
            break;
    }

    int[] indices = getScaleIndices();

    // Switch to flat notes for D and G minor scales
    if ((startIndex == 10 || startIndex == 5) && indices[2] == 3)
        notes = flatNotes;

    // Assign notes based on scale indices and the start index
    for (int i = 0; i < indices.Length; i++)
```

```
    {
        int index = startIndex + indices[i];
        if (index > 11)
            index -= 12;

        scale[i] = notes[index];
    }

    // If there are two notes with the same note name, replace one
    for (int i = 65; i <= 71; i++)  // For A to G
        if (containsTwoOf(scale, ((char)i).ToString()))
            replace(scale, ((char)i).ToString());

    // Catches the problem of getting D-flat and D or G-flat and G both in one scale
    if ((scale[0] == "D" || scale[0] == "G") && indices[2] == 3)
        replace(scale, notes[startIndex] + "\u266D");

    // Catches a similar problem with E-flat and A-flat minor scales
    if ((scale[0] == "E\u266D" || scale[0] == "A\u266D") && indices[2] == 3)
        replace(scale, notes[startIndex - 4]);

    return scale;
}
```

This method refers to two extraneous methods which I defined after writing this method. First, I created containsTwoOf, which returns true if a scale has two of one note name, such as A and A-flat.

```
        private bool containsTwoOf(string[] thisScale, string note)
        {
            bool flag1 = false;
            bool flag2 = false;

            for (int i = 0; i < thisScale.Length / 2; i++)
            {
                if (thisScale[i].Contains(note))
                {
                    if (!flag1)
                        flag1 = true;
                    else
                        flag2 = true;
                }
            }
            return flag1 && flag2;
        }
```

Next, I created the replace method. This method finds a note that needs replaced and replaces it with an equivalent note of a different name.

```
    private void replace(string[] thisScale, string note)
    {
        char[] noteChar = note.ToCharArray();
        switch (note)
        {
```

```
            // Replaces C with B-sharp, F with E-sharp, and so on
            case "C":
            case "F":
            case "D\u266D":
            case "G\u266D":
                for (int i = 0; i < thisScale.Length; i++)
                    if (thisScale[i] == note)
                        thisScale[i] = (((char)(noteChar[0] - 1)).ToString()) + "#";
                break;
            // Replaces B with C-flat and E with F-flat
            case "B":
            case "E":
                for (int i = 0; i < thisScale.Length; i++)
                    if (thisScale[i] == note)
                        thisScale[i] = ((char)(noteChar[0] + 1)).ToString() + "\u266D";
                break;
        }
    }
```

This concludes the implementation of my Scale class. I continued by implementing my four concrete strategy classes. These are all very similar and short. The code is as follows:

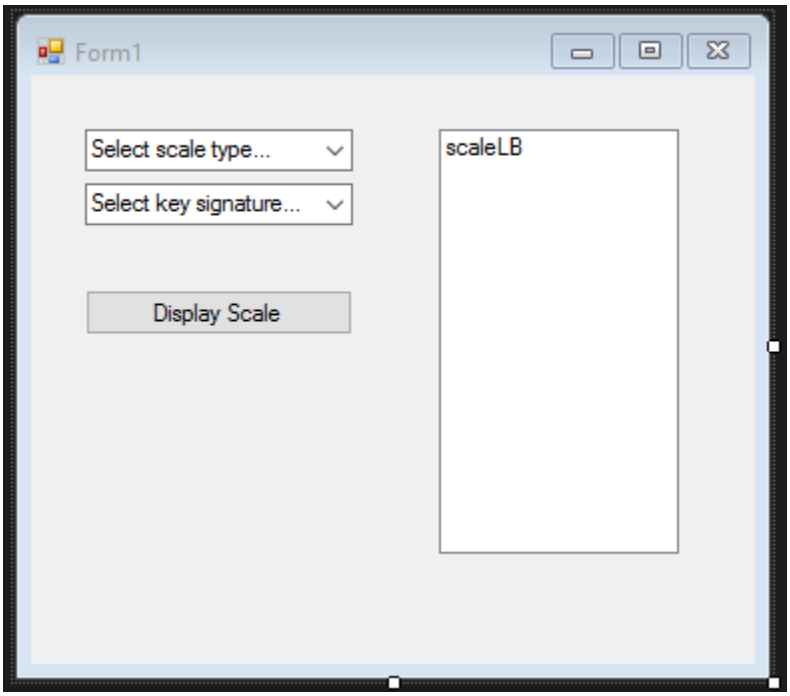
```
    class MajorScale : Scale   // This is a concrete strategy
    {
        public override int[] getScaleIndices()
        {
            int[] indices = { 0, 2, 4, 5, 7, 9, 11, 12, 11, 9, 7, 5, 4, 2, 0 };
            return indices;
        }

        public override string ToString()
        {
            return "Major scale";
        }
    }


    class NaturalMinorScale : Scale    // This is a concrete strategy
    {
        public override int[] getScaleIndices()
        {
            int[] indices = { 0, 2, 3, 5, 7, 8, 10, 12, 10, 8, 7, 5, 3, 2, 0 };
            return indices;
        }

        public override string ToString()
        {
            return "Natural minor scale";
        }
    }


    class HarmonicMinorScale : Scale // This is a concrete strategy
    {
        public override int[] getScaleIndices()
```

```
        {
            int[] indices = { 0, 2, 3, 5, 7, 8, 11, 12, 11, 8, 7, 5, 3, 2, 0 };
            return indices;
        }

        public override string ToString()
        {
            return "Harmonic minor scale";
        }
    }

    class MelodicMinorScale : Scale // This is a concrete strategy
    {
        public override int[] getScaleIndices()
        {
            int[] indices = { 0, 2, 3, 5, 7, 9, 11, 12, 10, 8, 7, 5, 3, 2, 0 };
            return indices;
        }

        public override string ToString()
        {
            return "Melodic minor scale";
        }
    }
```

The indices for these classes are hard-coded based on the definitions of each scale. With my concrete strategies implemented, I moved on to my Context class, called Pianist. This code is also very short and straightforward. I have two variables, a constructor, and a playScale method that calls getScale from the appropriate concrete class.

```
    class Pianist   // This is the Context class
    {
        Scale scale;    // Contains a scale of any type
        int startIndex; // Contains the start note, or key signature

        public Pianist(Scale scale, int startIndex)
        {
            this.scale = scale;
            this.startIndex = startIndex;
        }

        public string[] playScale()
        {
            string[] scaleNotes = scale.getScale(startIndex);
            return scaleNotes;
        }
    }
```

This code concluded the writing of my strategy class as described by the UML diagram. When I was done with this, I designed the form. A screenshot of the form from the design view of Visual Studio is shown below.



The user can select what scale type he wants from the top combo box (either major, natural minor, harmonic minor, or melodic minor), and can select the key signature from the second combo box. Then, when the user clicks the Display Scale button, the scale is displayed in the scale listbox.

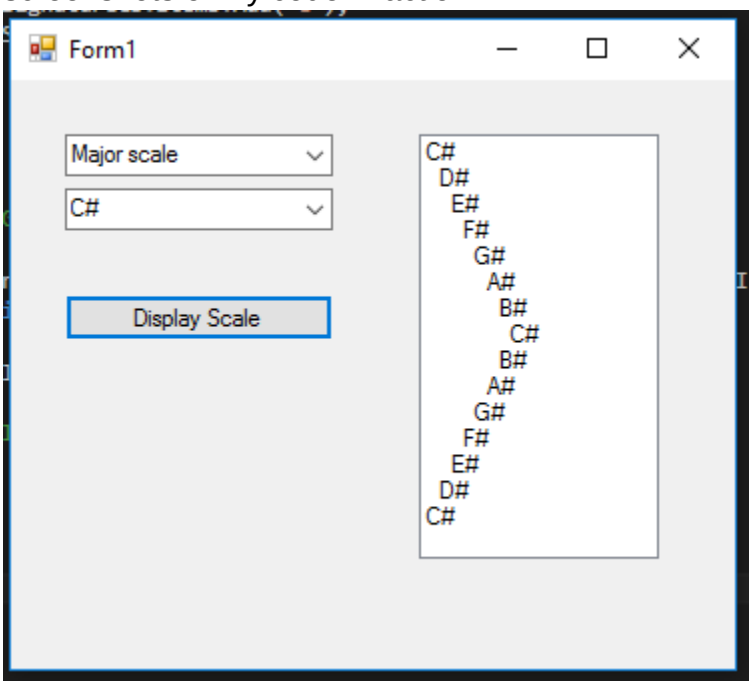
After I designed the form, I began writing code for it. I began by creating a Pianist variable and adding several items to the combo boxes.

```
Pianist pianist;

public Form1()
{
    InitializeComponent();

    scaleTypeCB.Items.Add(new MajorScale());
    scaleTypeCB.Items.Add(new NaturalMinorScale());
    scaleTypeCB.Items.Add(new HarmonicMinorScale());
    scaleTypeCB.Items.Add(new MelodicMinorScale());

    keySignatureCB.Items.Add("A");
    keySignatureCB.Items.Add("B\u266D");
    keySignatureCB.Items.Add("B");
    keySignatureCB.Items.Add("C");
    keySignatureCB.Items.Add("C#");
    keySignatureCB.Items.Add("D");
    keySignatureCB.Items.Add("E\u266D");
    keySignatureCB.Items.Add("E");
    keySignatureCB.Items.Add("F");
    keySignatureCB.Items.Add("F#");
    keySignatureCB.Items.Add("G");
    keySignatureCB.Items.Add("A\u266D");
}
```

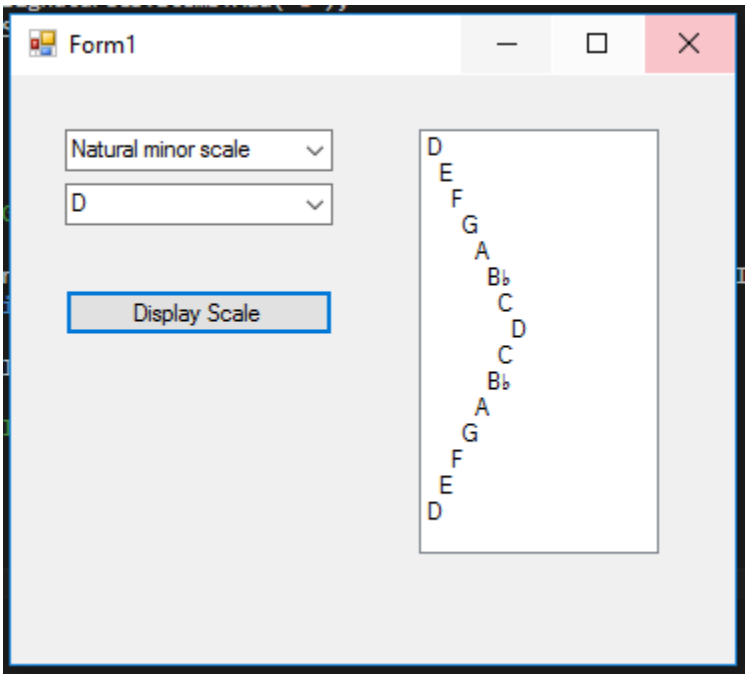
The only other code that I found necessary to write for the form was the click event method for the display button. The code is as follows:

```
private void displayButton_Click(object sender, EventArgs e)
{
    // Get the scale notes
    int startIndex = keySignatureCB.SelectedIndex;
    pianist = new Pianist((Scale)scaleTypeCB.SelectedItem, startIndex);
    string[] scaleNotes = pianist.playScale();

    scaleLB.Items.Clear();

    // Indent and display the notes
    for (int i = 0; i <= scaleNotes.Length / 2; i++)
    {
        string indent = "";
        for (int j = 0; j < i; j++)
            indent += "  ";

        scaleLB.Items.Add(indent + scaleNotes[i]);
    }

    // Descrease the indent on the way down
    int iDecrease = 2;
    for (int i = scaleNotes.Length / 2 + 1; i < scaleNotes.Length; i++)
    {
        string indent = "";
        for (int j = 0; j < i - iDecrease; j++)
            indent += "  ";

        scaleLB.Items.Add(indent + scaleNotes[i]);
        iDecrease += 2;
    }
}
```

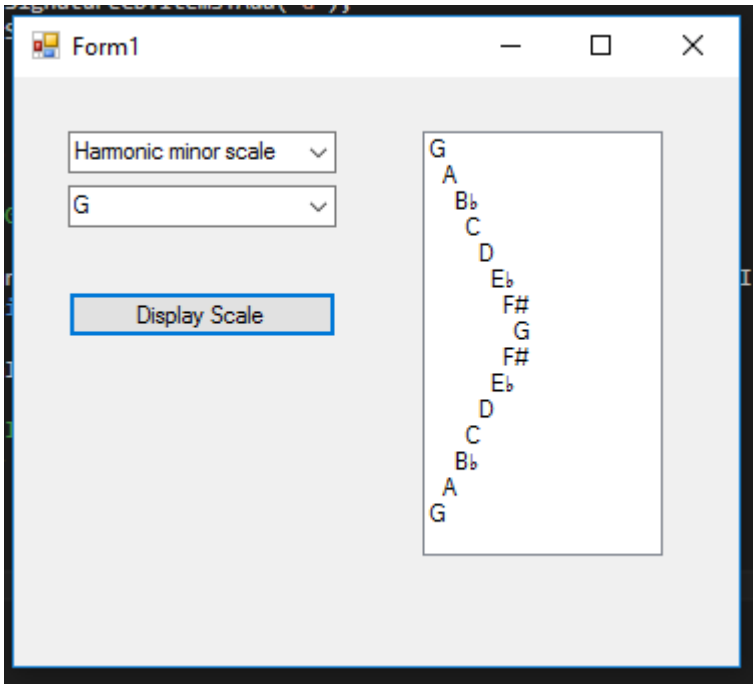
With this, I had finished writing the code for my application. Following are several screenshots of my code in action.



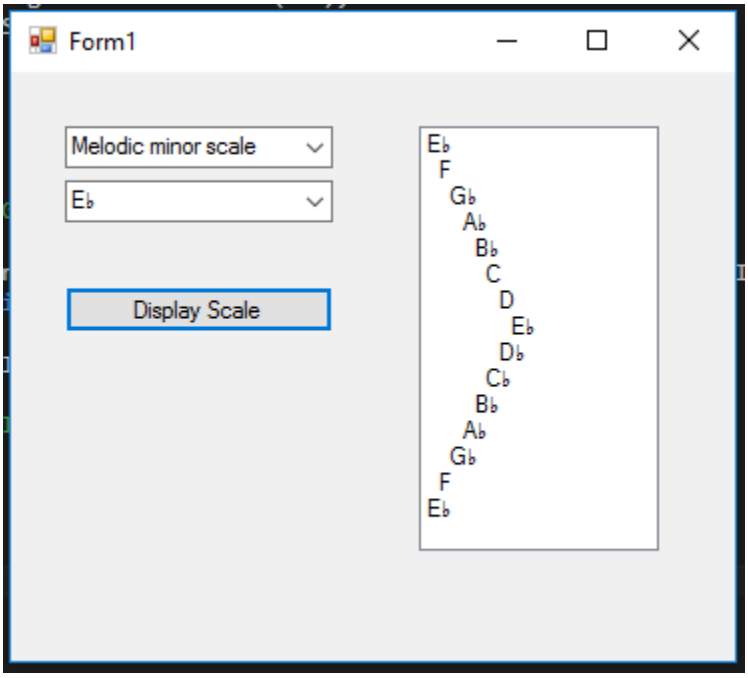
This screenshot shows the most relevant scale, C# major, displayed in the list box. This scale made use of the replace function in the Scale class.



Here, I displayed D minor, which is generally written only in sharps. However, due to the replace function, you can see the scale displayed properly with no sharps and one flat.



G harmonic minor demonstrates the same things shown in the D minor scale, but with a sharp shown as well.



Finally, I show E-flat melodic minor, which is different ascending and descending. These four screenshots show the functionality of all of the parts of my code described throughout this paper.

Observations

I found this pattern very easy to understand once I saw a demonstration in class. The Distance Finder program was, to me, the most helpful demonstration I've seen so far. Initially, I was unsure of the use of the Context class, but I believe that I implemented it correctly in my application. The use of actual objects in the combo box instead of string literals simplified my code a lot, and I will continue to do this in the future where it is appropriate. The most difficult part of this program was dealing with the exceptions to normal scale rules that appear with minor scales, such as the ability to have sharps and flats in one scale. I enjoyed writing this program because it could be useful in a music education context and it gave me experience with adapting to real-world problems such as variations in the rules that govern the implementation of different methods.