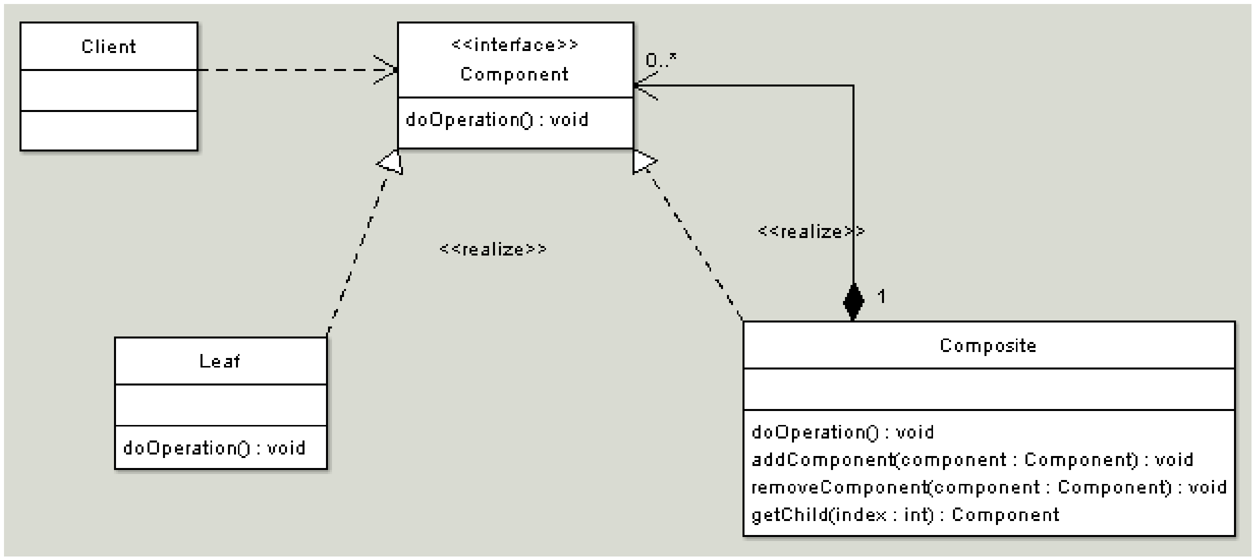
The Composite Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016-2017

Introduction

This assignment requires an application that makes use of the composite pattern. This pattern allows for the implementation of a tree structure. A Component interface is inherited from by the Leaf class and the Composite class. A Composite object can have children of either the Composite class or the Leaf class, while Leaf objects cannot have children. My application simulates an orchestra with four sections: the violins, the violas, the cellos, and the basses. The user can add one of each section to the orchestra and can add multiple musicians to each section. The user can also remove individual musicians or entire sections.

The UML Diagram



Above is the UML diagram for the composite pattern, courtesy of oodesign.com. As discussed in the introduction, the Component interface is inherited from by the Leaf class and the Composite class. The component interface has a function prototype for the operations that can be performed by both Leaf and Composite objects. The Leaf class implements only this method, while the Composite class implements this method as well as an add, remove, and getChild method. The Component interface is manipulated by the client. The table below describes the way I implement these classes in my program.

Client	The client in my program is the form, which manipulates a Composite object representing the orchestra that the user interacts with.
Component	My component interface is called Component. It provides a signature for the doOperation method, which in my case, gets the number of musicians in a Composite or Leaf object.
Composite	The Composite class in my program is called Section, and it represents a section of an orchestra. The possible sections are violins, violas, cellos, and basses.
Leaf	My Leaf class is called Musician. It represents an individual musician in one section of the orchestra.

Narrative and Code

I started writing my code by creating the Component interface, which was very simple.

```
interface Component // Component interface
{
    int getCount();
}
```

Then, I implemented the Musician class. I started by creating the variables that will be used throughout the implementation. I needed a name variable, a random number generator, and a string array containing several potential first names.

```
        string name = "";
        Random rngesus = new Random();
        string[] names = { "Andreya", "Andrew", "Ben", "Claire", "David", "Eugene",
                           "Francine", "Greg", "Harry", "Jaired", "Jacob", "Jack",
                           "Julie", "James", "Kevin", "Leonard", "Ophelia", "Penny",
                           "Rachel", "Steven", "Tori", "Violet", "William"};
```

I continued by creating two constructors; one has no arguments and randomly creates a name, while the other is passed a name.

```
        public Musician()
        {
            string first = names[rngesus.Next(names.Length)];
            char last = (char)rngesus.Next(65, 91);

            name = first + " " + last.ToString() + ".";
        }

        public Musician(string name)
        {
            this.name = name;
        }
```

Next, I implemented the getCount method. Since a musician is always a singular person, this method simply returns 1.

```
public int getCount()
{
    return 1;
}
```

I overrode the ToString method for use in displaying the musicians in the form:

```
public override string ToString()
{
    return "    " + name;
}
```

The spaces before the name are for the purpose of indentation in the form, which helps the user understand the tree structure being implemented. Finally, I overrode the Equals method for use in the remove method of the Section class.

```
public override bool Equals(Object obj)
{
    if (obj == null || this.GetType() != obj.GetType())
        return false;

    Musician m = (Musician)obj;
    return name == m.name;
}
```

Once I finished implementing the Musician class, I moved on to the Section class. I started by creating the two variables I would need for this class: a list of children and a name.

```
List<Component> children = new List<Component>();
string name = "";
```

I then implemented a constructor, which takes a section name as an argument, as well as the add, remove, getChild, and getCount methods.

```
public Section(string name)
{
    this.name = name;
}

public void addComponent(Component comp)
{
    children.Add(comp);
}
```

```
public void removeComponent(Component comp)
{
    if (children.Contains(comp))
        children.Remove(comp);
}


Component getChild(int index)
{
    Component[] childrenArray = children.ToArray();
    return childrenArray[index];
}

// Adds up the counts of all of the children of the section
public int getCount()
{
    int count = 0;
    for (int i = 0; i < children.Count; i++)
        count += getChild(i).getCount();

    return count;
}
```

Next, I overrode the ToString method, which returns the name of the section as well as the number of musicians in the section.

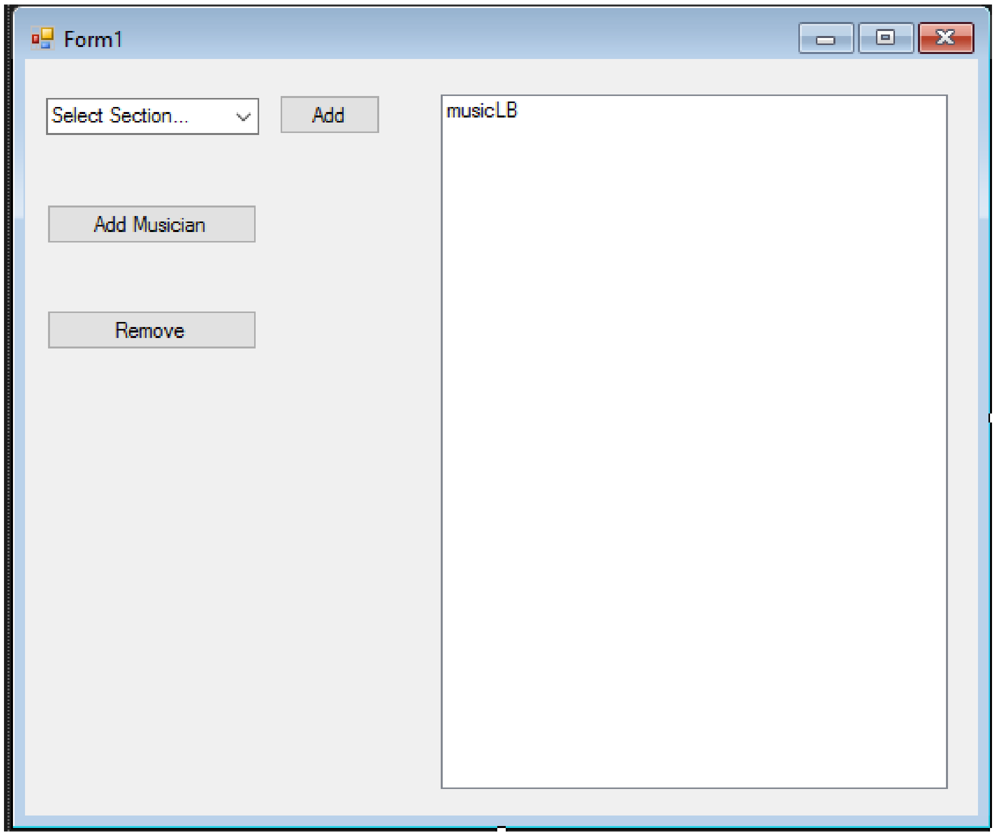
```
public override string ToString()
{
    return name + " (" + getCount() + ")";
}
```

Finally, I implemented a method called ToStringArray, which returns an array of strings for use in displaying the section in the form. Each string is either a section name with its musician count or a musician name. The code for this method is as follows:

```
public virtual string[] ToStringArray()
{
    string[] retval = new string[20 + this.getCount()]; // Add extra room for the
      lines that have section names
    retval[0] = name + " (" + getCount() + ")";         // Section name and count
    int currIndex = 1;

    // For each child
    for (int i = 0; i < children.Count; i++)
    {
        // If the child is a section
        if (getChild(i) is Section)
        {
            // Create a new string array to hold the section's information
            Section sec = (Section)getChild(i);
            string[] stringArray = new string[sec.getCount() + 1];

            // Set each string in the array to an empty string
            for (int l = 0; l < stringArray.Length; l++)
```

```
                    stringArray[l] = "";

                // Recursive call!
                stringArray = sec.ToStringArray();

                // Copy the new string array to the return array
                for (int j = 0; j < stringArray.Length; j++)
                {
                    if (stringArray[j] != null && !stringArray[j].Equals(""))
                    {
                        retval[currIndex] = "     " + stringArray[j]; // Spaces added
                          for visual clarity in the form
                        currIndex++;
                    }
                }
            }
            // If the child is a musician object, add the name to the array.
            else
            {
                retval[currIndex] = getChild(i).ToString();
                currIndex++;
            }
        }
        return retval;
    }
```

This concludes the implementation of my Section class. I finally moved on to designing and writing the code for the form. Below is a snip of my form in the design view:



The user can select one of the four sections listed earlier and add it to the orchestra, then add and remove musicians. The user can also remove entire sections from the orchestra.

After I designed the form, I began to implement the code for the form. I started by creating the variables I needed for the form and filling in the constructor.

```
Section myOrchestra;    // The variable for the whole orchestra
Section[] sections = new Section[5];    // 0 for orchestra; 1 for violins; 2 for
  violas; 3 for cellos; 4 for basses

 public Form1()
 {
     InitializeComponent();

     myOrchestra = new Section("My Orchestra");

     // Add the sections to the combo box for selecting sections
     sectionCB.Items.Add("Violins");
     sectionCB.Items.Add("Violas");
     sectionCB.Items.Add("Cellos");
     sectionCB.Items.Add("Basses");

     sections[0] = myOrchestra;

     updateLB();
 }
```

I created a function called updateLB, which was called in the constructor; this function displays the orchestra in the listbox.

```
private void updateLB()
 {
     // Reset the listbox and get the string array to add
     musicLB.Items.Clear();
     string[] stringArray = myOrchestra.ToStringArray();

     // For each element in the array
     for (int i = 0; i < 20 + myOrchestra.getCount(); i++)
     {
         // If the item is empty, break
         if (stringArray[i] == null)
             break;

         // Otherwise, add the item to the listbox
         musicLB.Items.Add(stringArray[i]);
     }
 }
```

I then started to write the code for each button click event. I started with the addSectionButton event. If the section is not already present in the orchestra, the function adds a new section to the orchestra and to the sections array and updates the listbox.

```
        private void addSectionButton_Click(object sender, EventArgs e)
        {
            // If the section is not already in the orchestra
            if (sections[sectionCB.SelectedIndex + 1] == null)
            {
                // Add the section to the array and the orchestra
                string secName = sectionCB.GetItemText(sectionCB.SelectedItem);
                Section sec = new Section(secName);
                myOrchestra.addComponent(sec);
                sections[sectionCB.SelectedIndex + 1] = sec;
            }
            // Otherwise, tell the user they can't do that.
            else
                MessageBox.Show("You already have one of those.");

            updateLB();
        }
```

Next, I implemented the addMusicianButton click event, which adds a musician to the appropriate section based on the index of the selected item in the listbox. It then updates the listbox.

```
    private void addMusicianButton_Click(object sender, EventArgs e)
    {
        int index = musicLB.SelectedIndex;

        if (musicLB.GetItemText(musicLB.SelectedItem).Contains("Violins"))
            sections[1].addComponent(new Musician());
        else if (musicLB.GetItemText(musicLB.SelectedItem).Contains("Violas"))
            sections[2].addComponent(new Musician());
        else if (musicLB.GetItemText(musicLB.SelectedItem).Contains("Cellos"))
           sections[3].addComponent(new Musician());
        else if (musicLB.GetItemText(musicLB.SelectedItem).Contains("Basses"))
            sections[4].addComponent(new Musician());

        else
            MessageBox.Show("Select a section to add the musician to.");

        // Update the listbox and set the selected index to the previously selected index
        updateLB();
        musicLB.SelectedIndex = index;
    }
```

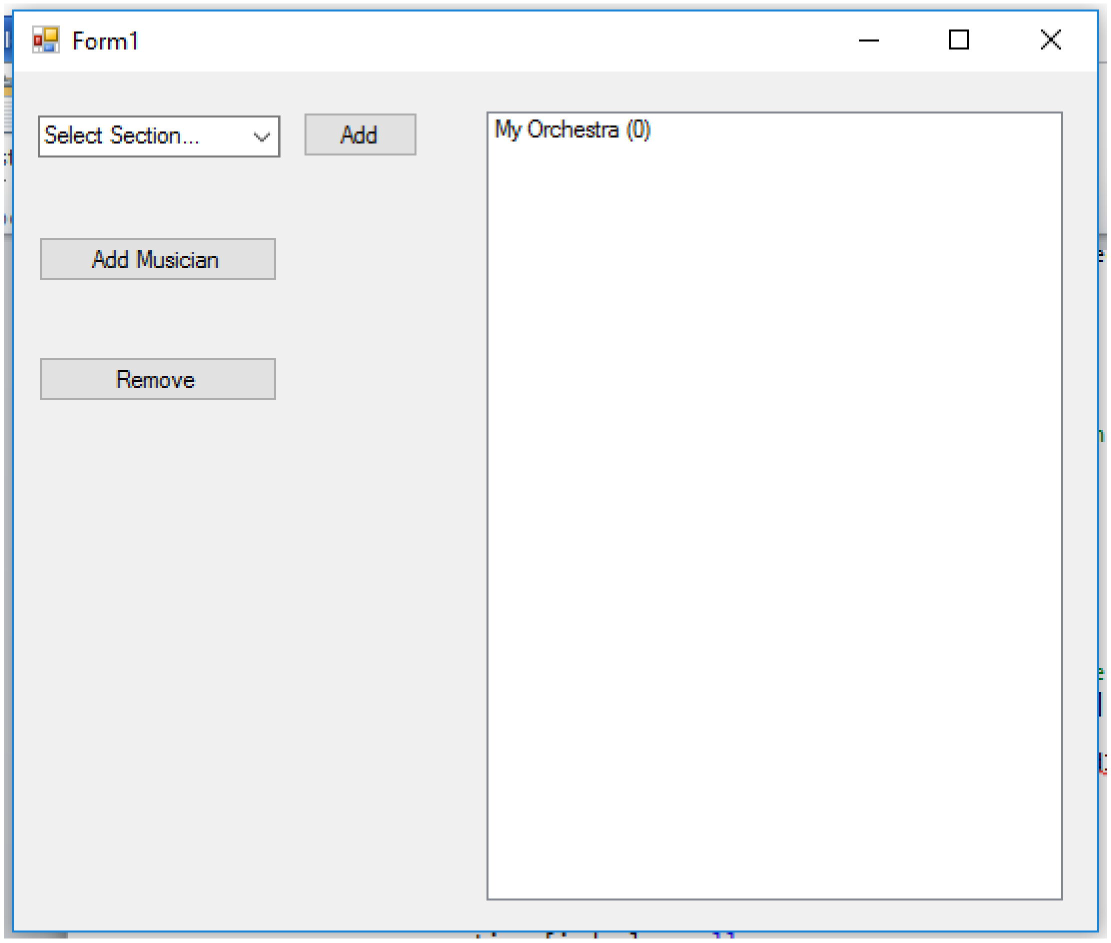
Finally, I implemented the removeButton click event. Based on the index of the selected item in the listbox, this function removes either the appropriate musician or the appropriate section, then updates the listbox.

```
private void removeButton_Click(object sender, EventArgs e)
{
    int indexToRemove = musicLB.SelectedIndex;

    int[] indices = new int[5]; // Index of the violin, viola, cello, and bass labels in
       the LB

    // Default the indices to -1.
    for (int i = 0; i < indices.Length; i++)
        indices[i] = -1;

    // If the section is present in the orchestra, set the index to the position of the
       section label in the listbox
    if (sections[4] != null)
        indices[4] = musicLB.FindString("    Basses (" + sections[4].getCount() + ")");

    if (sections[3] != null)
        indices[3] = musicLB.FindString("    Cellos (" + sections[3].getCount() + ")");

    if (sections[2] != null)
        indices[2] = musicLB.FindString("    Violas (" + sections[2].getCount() + ")");

    if (sections[1] != null)
        indices[1] = musicLB.FindString("    Violins (" + sections[1].getCount() + ")");

    // For each section in the array
    for (int i = 0; i < sections.Length - 1; i++)
    {
        // Start with the last section in the listbox
        // Get this index by finding the max index listed in indices.
        int num = indices.Max();
        int index = -1;

        for (int j = 0; j < indices.Length; j++)
            if (indices[j] == num)
                index = j;

        // If a musician in this section is selected, remove that musician
        if (indexToRemove > indices[index] && indices[index] != -1)
            sections[index].removeComponent(new
              Musician((musicLB.GetItemText(musicLB.SelectedItem)).Trim()));

        // If the section is selected, remove the section
        else if (indexToRemove == indices[index])
        {
            myOrchestra.removeComponent(sections[index]);
            sections[index] = null;
        }

        // Change this index to -2 so it isn't checked again
        indices[index] = -2;
    }
    updateLB();
}
```

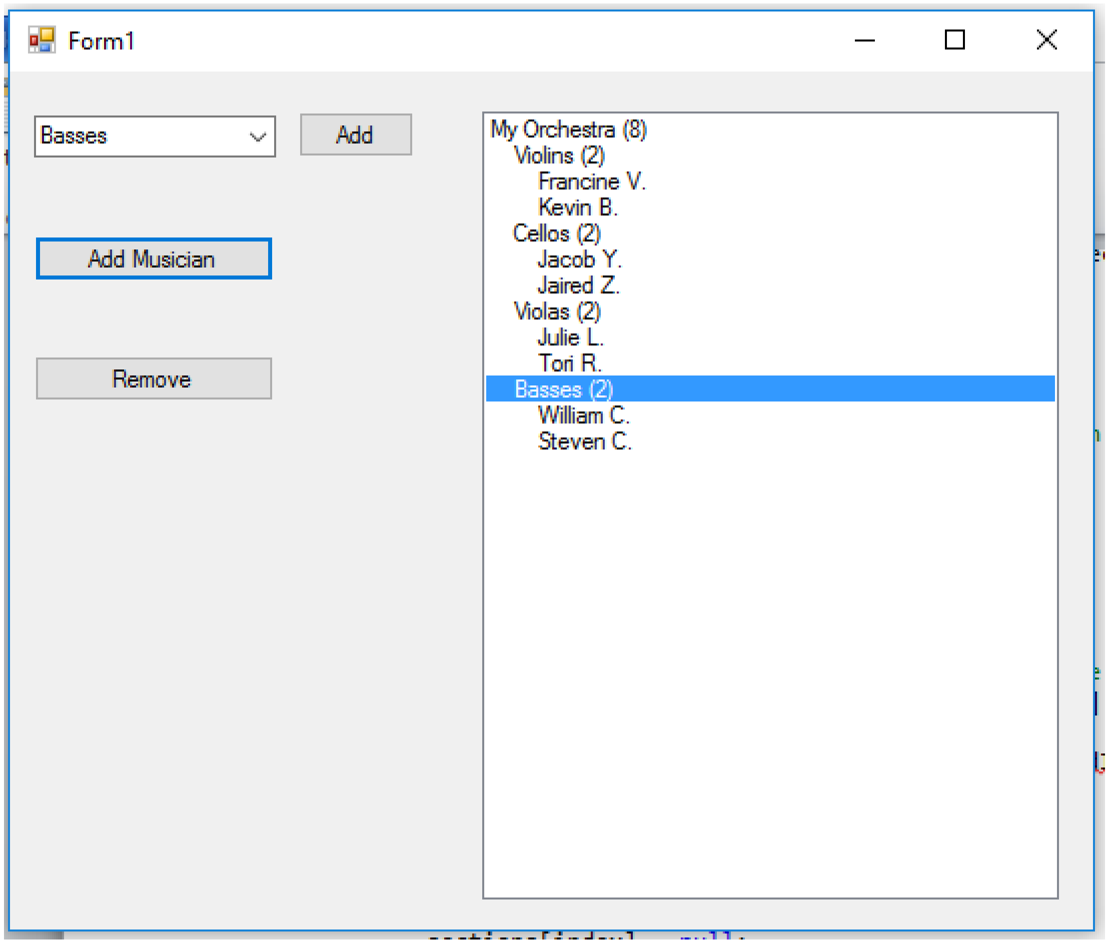
This concludes the code I wrote for this program.

The Deliverable

Below are several screenshots of my program in action.

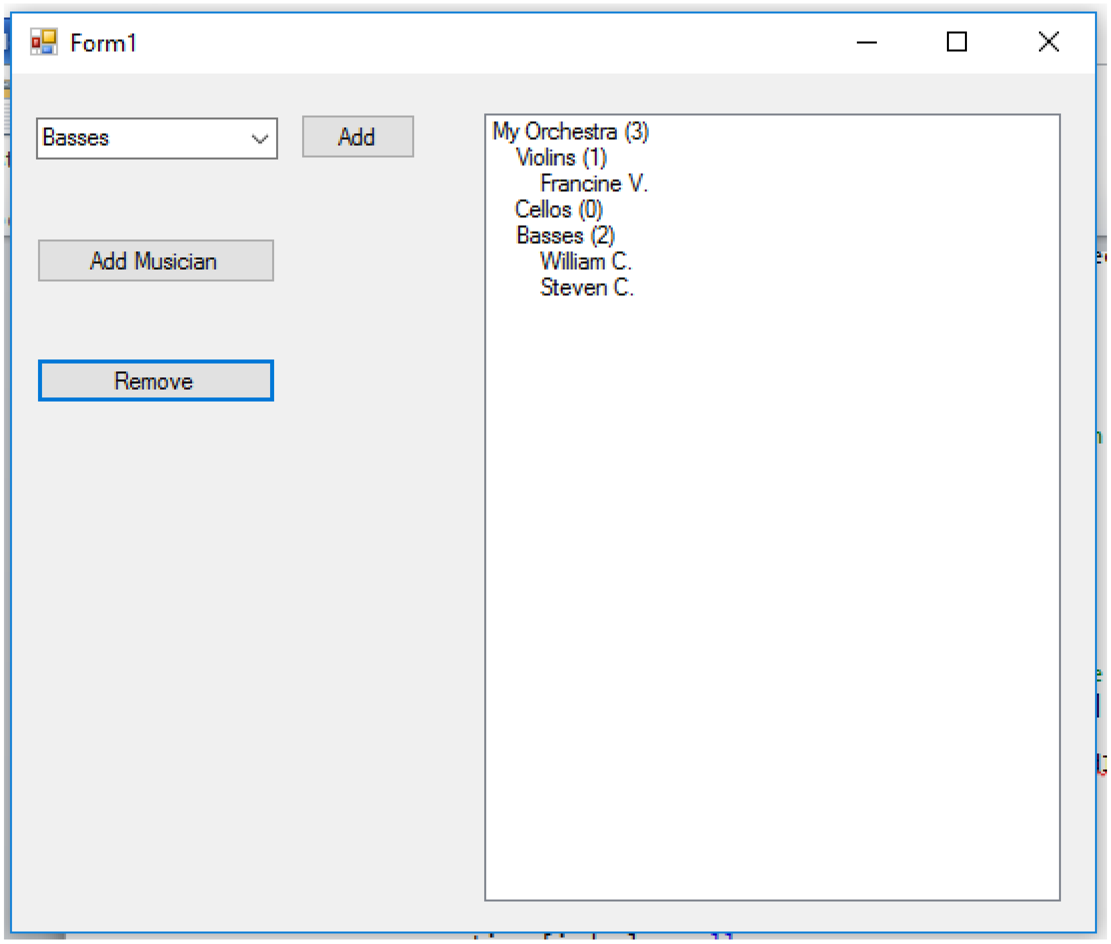


This is the form upon starting my application. The orchestra has no sections and no members.



```
My Orchestra (8)
  Violins (2)
    Francine V.
    Kevin B.
  Cellos (2)
    Jacob Y.
    Jaired Z.
  Violas (2)
    Julie L.
    Tori R.
  Basses (2)
    William C.
    Steven C.
```

I added each section to the orchestra (out of order) and then added two musicians to each section.



```
My Orchestra (3)
    Violins (1)
        Francine V.
    Cellos (0)
    Basses (2)
        William C.
        Steven C.
```

Finally, I removed some musicians and the viola section. The orchestra count has been updated appropriately.

<u>Observations</u>

This pattern was very interesting to write because I have experience using it from every time I've used a file directory system. I had a lot of problems with my ToStringArray method, but the code for the actual pattern (such as the add, remove, and getChild methods along with the manipulation of Section and Musician objects) was easy to write. After my intense struggles with getting my orchestra to display in the listbox properly, I was very excited to see my program running properly.