Jackson Spencer

The Iterator Pattern

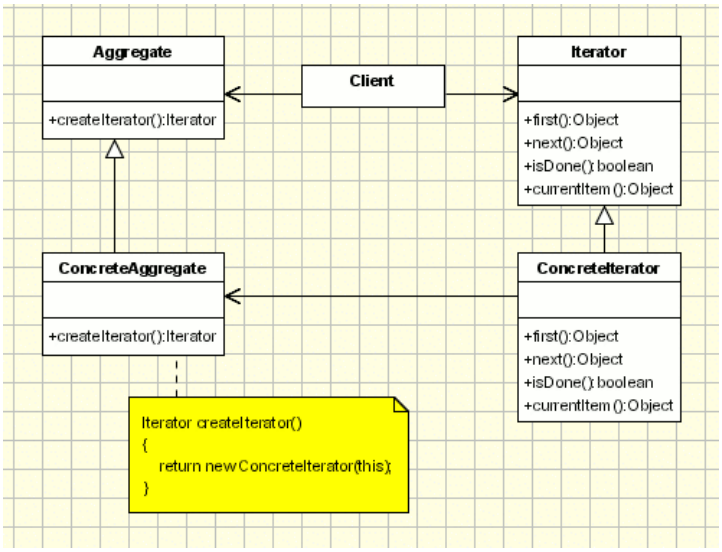
September 2, 2016

ECCS 2411: Design Patterns

Introduction

The assignment required an application to demonstrate the Iterator pattern. We were to implement and use the design pattern using a set of data to be used in an array, a list, queue, or stack. By definition, the Iterator pattern requests a way to "access the elements of an aggregate object sequentially without exposing its underlying representation." This submission used a list comprised of the finalist results of the 200 Meter Medley race from the Rio Summer Olympics of 2016.

UML Diagram



This UML diagram shows the components for each class and object required to implement the design pattern. The Aggregate and Iterator classes are both abstract. Another thing that can be seen here is that none of the classes can be started without the Iterator class, which is why that is the first class to generate. I have created another class that is not shown in the UML, but I will describe later. Below I have listed what each part of the UML diagram contains in my application.

The iterator class is abstract, therefore I created abstract methods to be used by the iterator class in the ConcreteIterator and the ReverseConcreteIterator classes.

```
public abstract class Iterator // this is the iterator class
    {
        public abstract object First();
        public abstract object Next();
        public abstract bool IsDone();
        public abstract object currentItem();
```

The iterator class was created first because it was the only class that did not rely on another in the UML diagram. The next class I created was the Aggregate class shown below.

```
public abstract class Aggregate // the Aggregate class
    {
        public List<string> Swimmers;
        public abstract Iterator createIterator();
        public abstract Iterator createReverseIter();
    }
```

The aggregate class is also abstract, and is inherited by the ConcreteAggregate class and is used in the form code. The createIterator() and createReverseIter() are public abstract methods to create iterators for a forward list and a backwards one. In the Aggregate, I declare that I am using a List to store the data of Swimmers. Because the ConcreteIterator class is implemented in another class, I created the class it was used in next, which was ConcreteAggregate. This class inherits from the Aggregate class, and it is used to gather the data to iterate in the form.

```
public class ConcreteAggregate : Aggregate // the Concrete Aggregate class
    {
        public ConcreteAggregate()
        {
            Swimmers = new List<string>();
        }

        public override Iterator createIterator()
        {
            return new ConcreteIterator(this);
        }

        public override Iterator createReverseIter()
        {
            return new ReverseConcreteIterator(this);
```

It creates the elements from my list of strings and gives the data to the iterators. The createIterator() method returns an instance of ConcreteIterator(this), returning ConcreteIterator using the aggregate's data. ReverseConcreteIterator does the same. There is a difference in how they iterate the data. However, they are using the same list. ConcreteIterator inherits the iterator class, and defines what each method is doing. For the ConcreteIterator, the list reads from top to bottom, so the gold medalist was listed on the top, then silver, etc. The First() method starts the array at position 0, then returns the item at the index. Next() method adds one to the index if the array does not reach the end. The IsDone() method checks if the array is at the end, then returns true if the index turns out to be greater than the number of elements.

```
public class ConcreteIterator : Iterator
    {
        Aggregate aggregate;
        int index;

        public ConcreteIterator(Aggregate agr)
        {
            aggregate = agr;
        }

        public override object First()
        {
            index = 0;
            return currentItem();
        }

        public override object Next()
        {
            if (!IsDone())
                index++;
            return currentItem();
        }

        public override bool IsDone()
        {
            return (index > aggregate.Swimmers.Count - 1);
        }

        public override object currentItem()
        {
            if (IsDone())
                return null;
            return aggregate.Swimmers[index];
        }
    }
```

The ReverseConcreteIter is similar to ConcreteIterator, only the list is read in reverse, i.e. bottom to top.

```
public class ReverseConcreteIterator : Iterator
    {
         Aggregate reverseAggregate;
        int index;

        public ReverseConcreteIterator(Aggregate agr)
        {
            reverseAggregate = agr;
        }

        public override object First()
        {
            index = reverseAggregate.Swimmers.Count - 1; // iterates from the bottom up
            return currentItem();
        }

        public override object Next()
        {
            if (!IsDone())
              index--; // continues in the reverse
            return currentItem();
        }

        public override bool IsDone()
        {
           return index < 0; // tells the reverse action to stop at the end
        }

        public override object currentItem()
        {
            if (IsDone())
                return null;
            return reverseAggregate.Swimmers[index];
        }
    }
```

With each class created, I added each to the form. Each element was added to the list holding the swimmer's name and their finishing place. The iterator and the reverse iterator are declared at the bottom.

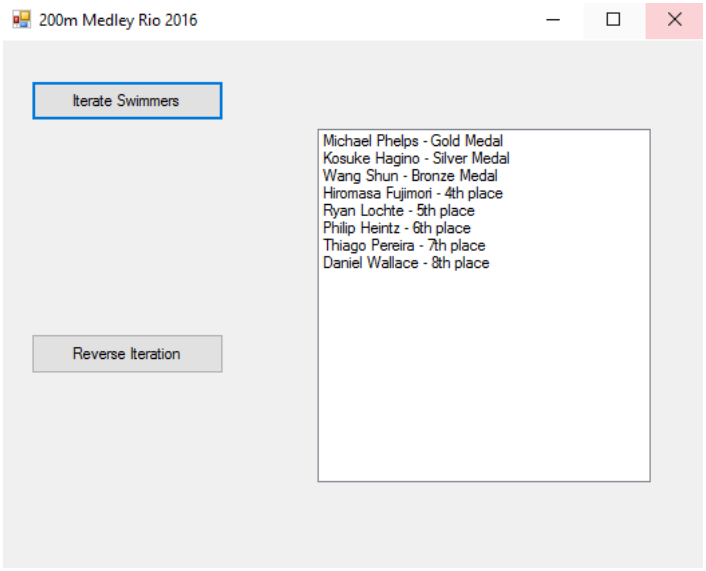
```
private void PrepareAgrWithIter()
        {
            agr.Swimmers.Add("Michael Phelps - Gold Medal");
            agr.Swimmers.Add("Kosuke Hagino - Silver Medal");
            agr.Swimmers.Add("Wang Shun - Bronze Medal");
            agr.Swimmers.Add("Hiromasa Fujimori - 4th place");
            agr.Swimmers.Add("Ryan Lochte - 5th place");
            agr.Swimmers.Add("Philip Heintz - 6th place");
            agr.Swimmers.Add("Thiago Pereira - 7th place");
```

```
            agr.Swimmers.Add("Daniel Wallace - 8th place");
            iterator = agr.createIterator();
            reverse = agr.createReverseIter();

               }
```

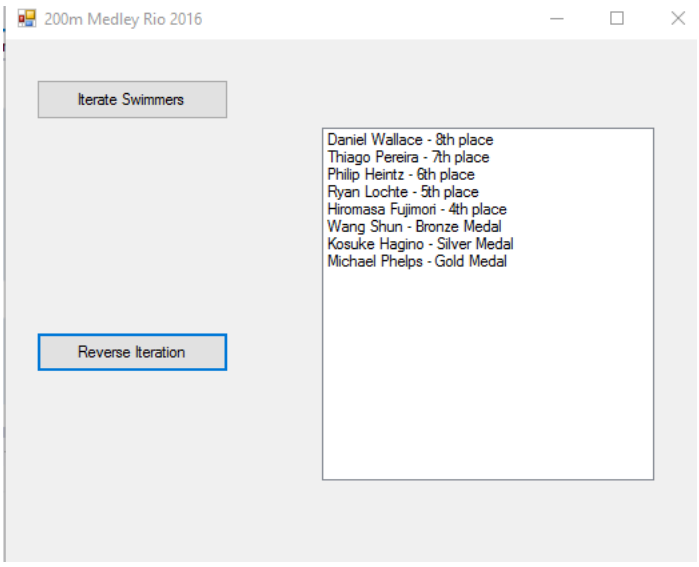
I created two separate buttons for the form; one to iterate normally and one to iterate the list in reverse. Each button adds their iterated list to the listbox.

```
private void btn_Iterate_Click(object sender, EventArgs e)
        {
            iterator.First();
            while (!iterator.IsDone())
            {
                lbx_Collection.Items.Add(iterator.currentItem());
                iterator.Next();
            }
        }

        private void btn_Reverse_Click(object sender, EventArgs e)
        {
            reverse.First();
            while (!reverse.IsDone())
            {
                lbx_Collection.Items.Add(reverse.currentItem());
                reverse.Next();
            }
               }
```

Iterate



Reverse



Reflection:

This was a great learning experience for me. This project initially confused me for most of the week we were learning about it. Watching the video was certainly beneficial to learning what each class did. I gained more experience in reading UML diagrams and how to use the C# language more effectively. I have had little experience with C# thus far, but I can already say that I prefer it over some other languages. The biggest part that gave me trouble was figuring out how to add the reverse iterator functionality. Once I decided to use a second button, I created a separate class to create a reverse concrete iterator. Implementing it into the form was the easy part. This assignment was a definite challenge, but a great learning experience at the same time. I feel that I have acquired a better understanding of the iterator design pattern.