Iterator Pattern

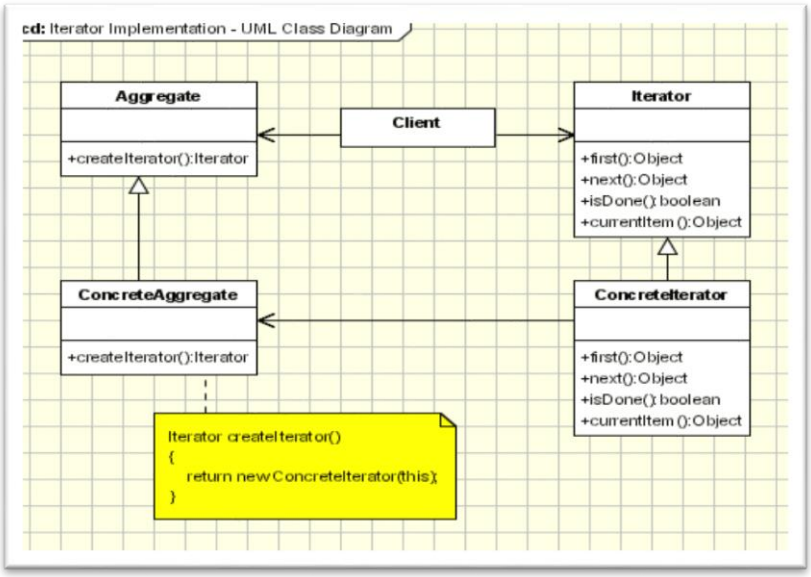
Design Patterns

Nathan Bemus

Iterator Pattern

Introduction

The purpose of this pattern is to access the contents of an aggregate object without exposing its underlying representation. This particular program demonstrates a very basic form of an iterator pattern that displays the names of my immediate family from oldest to youngest and vice versa. The program follows the following UML diagram.



UML Diagram Analysis

The diagram shows that the iterator pattern is divided into four classes. In it lies the Iterator Class, Aggregate Class, Concrete Iterator Class, and the Concrete Aggregate Class. The client accesses the Aggregate class and the Iterator class both of those classes use a concrete version of themselves. The Concrete Iterator also communicates with the Concrete Aggregate directly.

Program Walkthrough

Iterator Class

My specific class called Iterator looks like the following. The class instantiates a public abstract class called Iterator and defines abstract methods inside the class. According to

```
namespace Iterator_Pattern
{
    //This is the Iterator class
    public abstract class Iterator
    {
        public abstract object first();
        public abstract object next();
        public abstract bool isDone();
        public abstract object currItem();
    }
}
```

whatis.techtarget.com, abstract means to hide irrelevant data and increases efficiency.

```
namespace Iterator_Pattern
{
    //This is the Aggregate class
    public abstract class Aggregate
    {
        public List<string> Elements;

        public abstract Iterator createIterator();
        public abstract Iterator createReverseIterator();
    }
}
```

Aggregate Class

The next Class I created was the Aggregate class. This class contains the instantiation of the Elements object that is a list that is used to store strings (the names of my family) in this case. The methods defined are abstract as well and inherit from the Iterator class. These methods will be used in the Form1.cs class

Concrete Iterator

Afterwards I created the Concrete Iterator Class and the Reverse Iterator Class. These Two classes are very similar and use the same methods from the Iterator class. Here is a snippet of what the Concrete Iterator Class for my program looks like. The Concrete Iterator in this part starts off by declaring its inheritance to the Iterator Class. It then proceeds to declare a public method that will

```
namespace Iterator_Pattern
{
    public class ConcreteIterator : Iterator
    {
        int CurrIndex;
        Aggregate aggregate;

        public ConcreteIterator(Aggregate agg)
        {
            aggregate = agg;

        }

        public override object first()
        {
            CurrIndex = 0;
            return currItem();
        }

        public override object next()
        {
            if (!isDone())
            {
                CurrIndex++;
            }
            return currItem();
        }

        public override bool isDone()
        {
            return (CurrIndex > aggregate.Elements.Count - 1);
        }

        public override object currItem()
        {
            if (isDone())
            {
                return null;
            }
            return aggregate.Elements[CurrIndex];
        }
    }
}
```

be used later in the form. Next the Concrete Iterator gives meaning to the methods instantiated in the Iterator class.

Concrete Aggregate

This is the last class that I created and it's called the Concrete Aggregate Class. It creates the public method which allows the form to communicate with the Aggregate class and puts values in the previously stated list. It also creates methods that the form will use to access the Concrete Iterator and the Reverse Concrete Iterator.

```
namespace Iterator_Pattern
{
    //This is the ConcreteAggregate class
    public class ConcreteAggregate : Aggregate
    {
        public ConcreteAggregate()
        {
            Elements = new List<string>();
        }

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

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

Form1 Code (part 1)

This is the main part of the code where everything comes together and puts all the main information into the pattern. The following is the first half of the source code for Form1. In this half, an Aggregate object is declared and links to the Concrete Aggregate Class. Next I declare two Iterator objects that represent the Concrete Iterator Class and the Reverse Concrete Iterator Class. I then initialize all the components in the form and put string values in the list.

```
namespace Iterator_Pattern
{
    public partial class Form1 : Form
    {
        Aggregate agg = new ConcreteAggregate();
        Iterator iterator;
        Iterator reverse;

        public Form1()
        {
            InitializeComponent();
            PrepareAggWithIter();
        }

        private void PrepareAggWithIter()
        {
            agg.Elements.Add("Dave");
            agg.Elements.Add("Tammy");
            agg.Elements.Add("Tyler");
            agg.Elements.Add("Zach");
            agg.Elements.Add("Nathan");
            agg.Elements.Add("Luke");
            iterator = agg.createIterator();
            reverse = agg.createReverseIterator();
        }
```

Form1 Code Part2

This part is the functionality for the button in the form. The button clears out any previous strings from the list box and proceeds to go through the list like a broken up for loop. Based on the button selected it will traverse through the list forwards or backwards.

```
        private void btnIterate_Click(object sender, EventArgs e)
        {
            lbCollection.Items.Clear();
            iterator.first();
            while (!iterator.isDone())
            {
                lbCollection.Items.Add(iterator.currItem());
                iterator.next();
            }
        }

        private void btnReverseIterator_Click(object sender, EventArgs e)
        {
            lbCollection.Items.Clear();
            reverse.first();
            while (!reverse.isDone())
            {
                lbCollection.Items.Add(reverse.currItem());
                reverse.next();
            }
        }
    }
}
```

Reflection

This pattern uses four total classes that communicate with each other using inheritance to traverse through a list of strings. The pattern also gives the option to run through the list forwards and backwards.

Conclusion

This particular assignment taught me a basic understanding on how to read a UML diagram, how to make a basic Iterator Pattern, and was a refresher about working with C# OOP. I think the assignment was just the right difficulty for the first pattern and I am anxious to start working on the next one.