Jacob Scholl

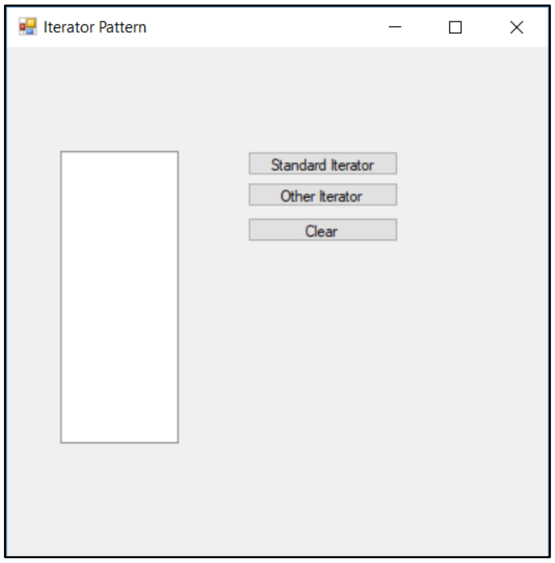
Prof. Retterer, Retired

Design Patterns

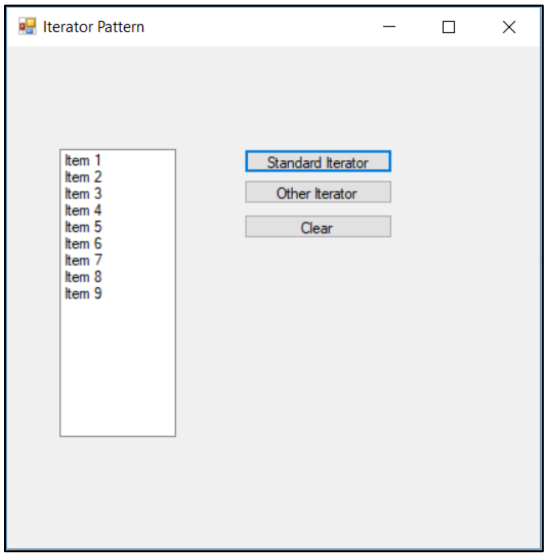
6 September 2017

Introduction: Do the iterator pattern.

Narrative: The iterator pattern is used for providing access to the elements in an aggregate sequentially without exposing its underlying implementation. This demo illustrates this functionality by using two iterators on the same aggregate.

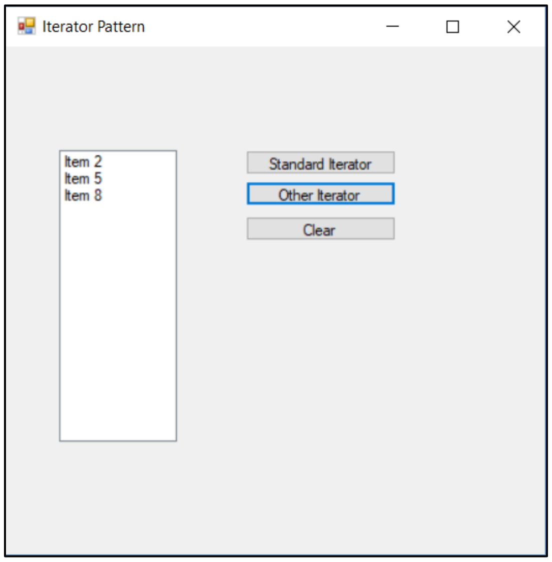


A list box of strings is displayed on the left side of the window, and three buttons control the data displayed inside that box. The first box fills the box with the data gathered by the first iterator, which iterates through all the elements in the aggregate object. In this demo, the aggregate object is a List that contains strings which can be accessed by index. This iterator finds the first string by getting the string at index 0 of the List and saving an index variable as 0. Each subsequent string is found by incrementing that variable by 1, thus eventually iterating through all strings in the aggregate. To check for the final string, the `Count()` function is called on the List, which returns the number of objects in the list. The returned value is then compared with the index variable to ensure that the iterator can access all objects in the aggregate without accidentally messing with other values in memory.

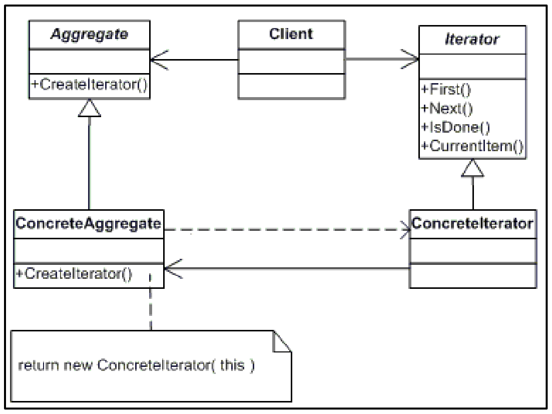


```
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
```

The second iterator, used in the second button, functions almost identically, but instead of iterating through every string in the List, the `Next()` function skips two strings at a time. Thus, it only displays every third string in the List. All other functions in that iterator act identically to the first one.



The third button simply clears the list box by calling `Items.Clear()`. This is a pre-defined functionality for a list box.



This UML diagram for an iterator shows the critical components of the pattern. Each of these components were utilized in my code:

Aggregate	I used a List<string>, which is pre-defined.
Iterator	I created an abstract class called Iterator that contains abstract functions First, Next, isDone, and CurrentItem.
Concrete Aggregate	The concrete aggregate inherits from the Aggregate class, thus in turn inheriting the functionality of the List<string>.
Concrete Iterator	Both the “standard” iterator and the “third” iterator are defined here. They both inherit the Iterator class, so the same functions are defined differently in both.
Client	The client is the demo application.

Although there are two different iterators featured in this demo, since they both inherit the same Iterator class (which follows the specifications of the Iterator pattern), they both access the same aggregates, and they are created and used in the same way, this demo still demonstrates the Iterator pattern.

Observations: This assignment seemed simple, rapidly became difficult, then finally settled back down to a manageable level. Originally, I was going to try to implement a binary tree as my aggregate, but the concept became too difficult once I attempted to implement an iterator that would work in the tree yet still follow the specifications of the UML diagram. I switched back to a much simpler list, and that proved to be a much more comprehensible project. Overall, this project was difficult, but understandably so.

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace IteratorPattern
{
    public partial class clientForm : Form
    {
        Aggregate agg = new ConcreteAggregate();

        public clientForm()
        {
            InitializeComponent();

            agg.Add("Item 1");
            agg.Add("Item 2");
            agg.Add("Item 3");
            agg.Add("Item 4");
            agg.Add("Item 5");
            agg.Add("Item 6");
            agg.Add("Item 7");
            agg.Add("Item 8");
            agg.Add("Item 9");
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }
        private void standardButton_Click(object sender, EventArgs e)
        {
            fullListBox.Items.Clear();

            Iterator iter = new ConcreteIterator(agg);
            for (iter.First(); !iter.IsDone(); iter.Next())
            {
                fullListBox.Items.Add(iter.CurrentItem());
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            fullListBox.Items.Clear();
        }

        private void otherIteratorButton_Click(object sender, EventArgs e)
        {
            fullListBox.Items.Clear();

            Iterator iter = new OtherIterator(agg);
            for (iter.First(); !iter.IsDone(); iter.Next())
            {
                fullListBox.Items.Add(iter.CurrentItem());
            }
        }
    }
}
```

Iterator.cs

```
namespace IteratorPattern
{
    public abstract class Iterator
    {
        protected Aggregate aggregate;

        public abstract void First();
        public abstract void Next();
        public abstract bool IsDone();
        public abstract string CurrentItem();
    }
}
```

ConcreteIterator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace IteratorPattern
{
    public class ConcreteIterator : Iterator
    {
        int index;

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

        public override string CurrentItem()
        {
            return aggregate[index];
        }

        public override void First()
        {
            index = 0;
        }

        public override bool IsDone()
        {
            if (index >= aggregate.Count())
            {
                return true;
            }
            return false;
        }

        public override void Next()
        {
            index++;
        }
    }

    public class OtherIterator : Iterator
    {
        int index;

        public OtherIterator(Aggregate agg)
        {
            aggregate = agg;
        }

        public override string CurrentItem()
        {
            return aggregate[index];
        }

        public override void First()
        {
            index = 1;
        }

        public override bool IsDone()
        {
            if (index >= aggregate.Count())
            {
                return true;
            }
            return false;
        }

        public override void Next()
        {
            index += 3;
        }
    }
}
```

Aggregate.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace IteratorPattern
{
    public abstract class Aggregate : List<string>
    {
        public abstract Iterator CreateIterator();
    }
}
```

ConcreteAggregate.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace IteratorPattern
{
    public class ConcreteAggregate : Aggregate
    {
        public override Iterator CreateIterator()
        {
            return new ConcreteIterator(this);
        }

        public Iterator CreateOtherIterator()
        {
            return new OtherIterator(this);
        }
    }
}
```