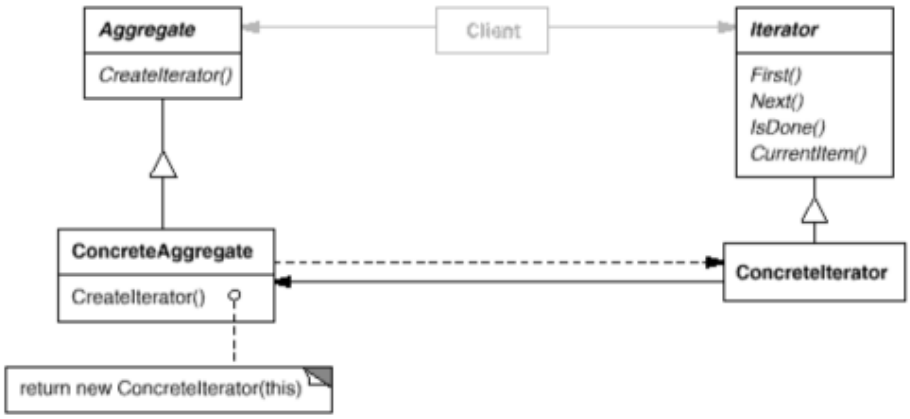
The Iterator Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016-2017

Introduction

This assignment requires an application that makes use of the Iterator Pattern. The Iterator Pattern is used to access the elements of an aggregate object sequentially without exposing the type of data structure used. This submission iterates through a list of names alphabetically, forward and backward, by first name.

The UML Diagram for Iterator

The UML diagram for this pattern is shown to the right. There are two abstract classes, Aggregate and Iterator. ConcreteAggregate and ConcreteIterator derive from Aggregate and Iterator, respectively. ConcreteAggregate is used to describe the type of data being traversed through, and ConcreteIterator implements the methods used to traverse the data. The table below summarizes the classes I used in my application to implement the Iterator Pattern.



Aggregate	The List structure is used in the abstract *Aggregate* class.
Iterator	The abstract class *Iterator* provides prototypes for the four abstract functions listed in the UML diagram under *Iterator*. The classes that act as ConcreteIterators implement those functions.
ConcreteAggregate	The ConcreteAggregate class derives from *Aggregate* and implements the list by using a list of strings. Each string is the first and last name of a student.
ConcreteIterator	My ConcreteIterator classes, ConcreteIterator and BackwardIterator, derive from *Iterator*. They are created in ConcreteAggregate and they traverse through the list of names forward and backward, respectively.
Client	The application is the client.

Narrative of Code

The abstract classes appear in my code exactly as they do in the Gang of Four's UML diagram. For example, the entirety of the Iterator class is as follows:

```
public abstract string first();
public abstract string next();
public abstract bool isDone();
public abstract string currentItem();
```

The implementations of these functions in the ConcreteIterator and BackwardIterator classes are where my code begins to become unique. The next() function is used the most frequently when iterating through my list of names. The function, shown below, selects the string from the list that is next alphabetically. A variable containing the previous item is created, and the next item in line is assumed to be the last item alphabetically. If we have already iterated through the entire list, the last item in the list, alphabetically, is returned. Otherwise, we enter a while loop. In the loop, two comparisons are made; if the current item comes after the previous item but before the item currently listed as the next item, then the next item and item index variables are updated with the current item's data. After the loop has terminated, we are left with our new current item, which we return.

```
public override string next()
{
    string prevItem = currentItem();
    string nextItem = last();
    int i = 0;
    numItemsIterated++;

    itemIndex = 0;

    if (numItemsIterated == aggregate.studentNames.Count - 1)
    {
        item = last();
        return currentItem();
    }

    while (i < aggregate.studentNames.Count)
    {
        if (string.Compare(aggregate.studentNames[i], prevItem) > 0 &&
            string.Compare(aggregate.studentNames[i], nextItem) < 0)
        {
            nextItem = aggregate.studentNames[i];
            itemIndex = i;
        }
        i++;
    }

    item = nextItem;
    return currentItem();
}
```

In order to iterate backwards, the greater than and less than signs must be reversed. No other changes are necessary.

```
namesListBox.Items.Clear();

iter.first();
while (!iter.isDone())
{
    namesListBox.Items.Add(iter.currentItem());
    iter.next();
}
```

Now that we can iterate forward and backward through the list, we need to display the results to the list box. This is done simply, as shown in the box to the left. It uses the operations defined in the abstract Iterator class to traverse through the list without exposing what type of list it is traversing through. Until the iterator has passed through all items, the current item is added to the list box. This simple function shows the user a list of names alphabetically. A very similar function is used to show the names reversed.

The screenshots on the following page illustrate the application's iteration functions.

Iterates forward through the names

Iterates backward through the names

Observations

This application could become useful if the ConcreteAggregate class was replaced by a Person class, with several attributes about each person provided and many different ways to sort the people (such as by age, by birthday, and so on). I would like to hear in class about how to create and use different types of iterators in one program, because I'm not sure if my implementation of different iterators follows the pattern or not. However, I think that the rest of my application meets the requirements, and this assignment taught me a lot about the behind-the-scenes work that loops and aggregate operations do.