Iterator Pattern

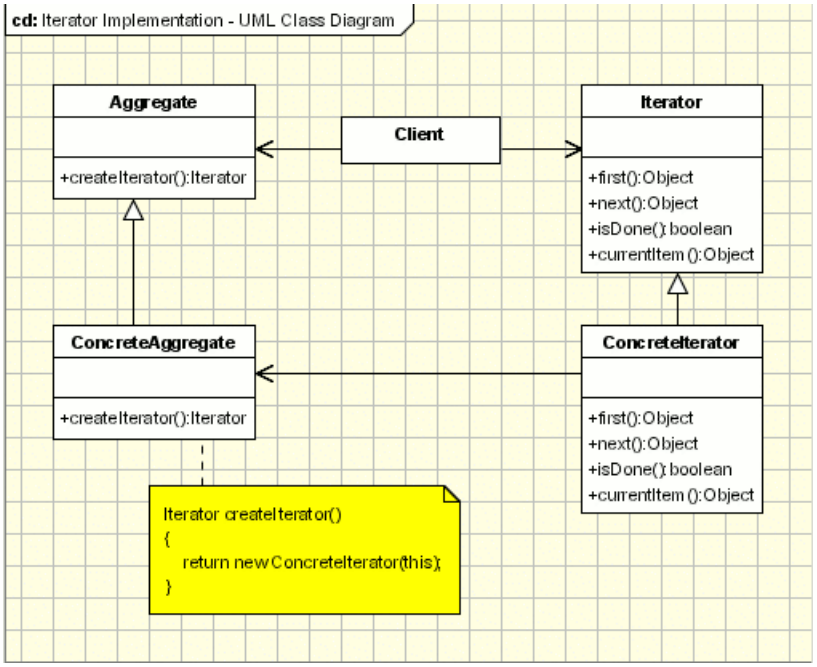
Design Patterns

Christopher Doseck

9/2/2016

Introduction

This assignment is an application that I created that shows how the Iterator Pattern works. In this application, I am using a List to store objects of the Team class. Each Team object has a team name, rank, wins, losses, and division.



The UML Diagram for Iterator

The UML diagram for the iterator pattern, shown on the right, shows the classes needed to have the requirements. I used the Aggregate and Iterator classes as abstract classes, and used children classes to inherit from them. The table below shows how each of the classes were used.

Aggregate	I used the List to be the aggregate in this assignment.
Iterator	The iterator class is an abstract class that has all of the methods that will be used in the children of this class.
Concrete Aggregate	This class is the child class of Aggregate. It is passed a string to determine which Iterator to set up when the create Iterator function is called, and populates the teams into itself when the add function is called.
East Iterator	This class is a child class of Iterator. It iterates through all of the teams that have their division as “Big Ten East”. It gets the aggregate passed to it when its constructor is called in the Concrete Aggregate.

West Iterator	This class is a child class of Iterator. It iterates through all of the teams that have their division as "Big Ten West". It gets the aggregate passed to it when its constructor is called in the Concrete Aggregate.
Rank Iterator	This class is a child class of Iterator. It iterates through all of the teams that have their ranking as a number. It gets the aggregate passed to it when its constructor is called in the Concrete Aggregate.
Winning Iterator	This class is a child class of Iterator. It iterates through all of the teams that have more wins than losses. It gets the aggregate passed to it when its constructor is called in the Concrete Aggregate.
Team	This is the class that has objects of it stored as an aggregate. It has a name, rank, wins, losses, and division.
Client	The demonstration application.

Narrative

The abstract **Iterator** class had this basic setup

```
public abstract class Iterator{
    public abstract Team first();
    public abstract Team next();
    public abstract Team currentItem();
    public abstract bool isDone();
}
```

The abstract **Aggregate** class had this basic setup

```
public abstract class Aggregate{
    public List<Team> elements;
    public abstract void add(Team team);
    public abstract Iterator createIterator(string iterType);
}
```

Concrete Aggregate

```
public override Iterator createIterator(string iterType){
    if (iterType == "West")
        return new WestIterator(this);
    else if (iterType == "East")
        return new EastIterator(this);
    else if (iterType == "Ranked")
        return new RankedIterator(this)
    else if (iterType == "Winning")
        return new WinningIterator(this
    else return null;
}
```

This code is used to see which of the children of the Iterator Class is being used, and then sends itself to that Iterator.

The other function in this class "add(Team team)" simply adds a Team object to the List of Teams already in the class.

East Iterator

```
public EastIterator(ConcreteAggregate agg)
{
    this.agg = agg;
}
```

This class starts off in the with assigning the ConcreteAggregate passed to it to a ConcreteAggregate in its own class

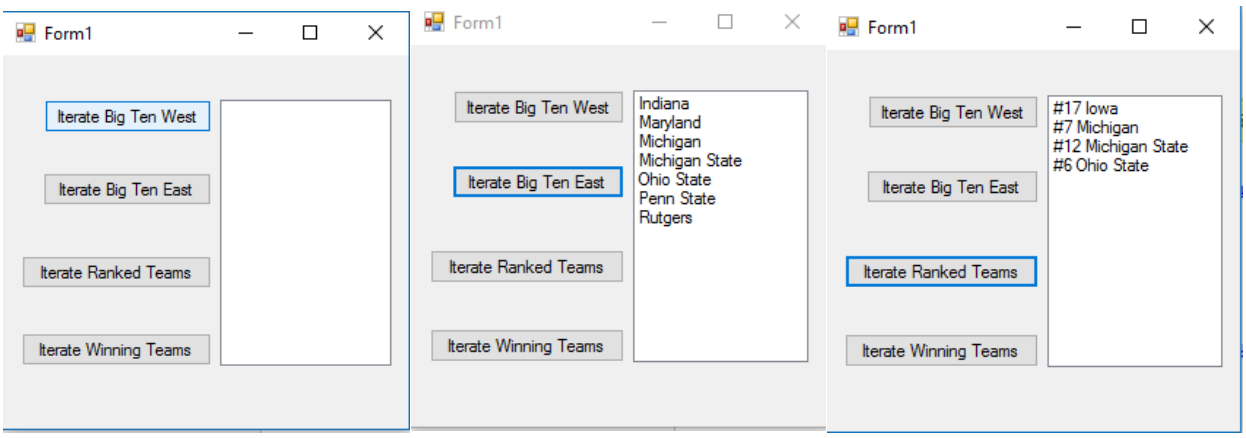
```
public override Team first(){
    itemNumber = 0;
    while (!("Big Ten East" == agg.elements[itemNumber].getDivision()))
        itemNumber++;
    return currentItem();
}

public override bool isDone()
{
    return itemNumber == agg.elements.Count;
}

public override Team currentItem()
{
    if (isDone())
       return null;
    return agg.elements[itemNumber];
}

public override Team next()
{
    itemNumber++;
    if (!isDone())
    {
        while (!("Big Ten East" == agg.elements[itemNumber].getDivision()))
        {
            itemNumber++;
            if (isDone())
               return null;
        }
    }
    return currentItem();
}
```

The first method sets the initial item number to be 0. It will then go check if the Team at the item number has the division of "Big Ten East". If it does, it will return the Team, otherwise it will increase the item number. Once it finds the first element with the division "Big Ten East" it will call currentItem()

isDone() checks to see if the current item number is the last number of the List

currentItem() checks to see if the List is done, and if it isn't, it returns the Team

next() gets the next item in the iteration. It starts out by going to the very next item, regardless of if it's wanted or not. Then it checks whether or not it is at the end of the aggregate. If it is not, it will check if the division is the "Big Ten East" or not. If it is not, it will add one to the item number and check if it is done. It will continue to do this until it is done, or it finds the next "Big Ten East" element. Once that is done, it will return the current item.



One with nothing run	One with IterateEast run	IterateRanked run

Observations

It was rather easy to get the abstract classes of Iterator and Aggregate to work. Getting there to be separate Iterators for each different part was difficult while making sure that I did not need a different ConcreteAggregate class for each Iterator, because I was using the same data for each Iterator. Another one of the hard parts was making sure that the Iterator stopped at the right time. When we were first assigned this pattern, I thought that it would be useless because we already had a way to iterate through things. After this, I realized how nice it is to have your own iterator to specify what gets iterated through. This project was challenging, but it was able to be done.