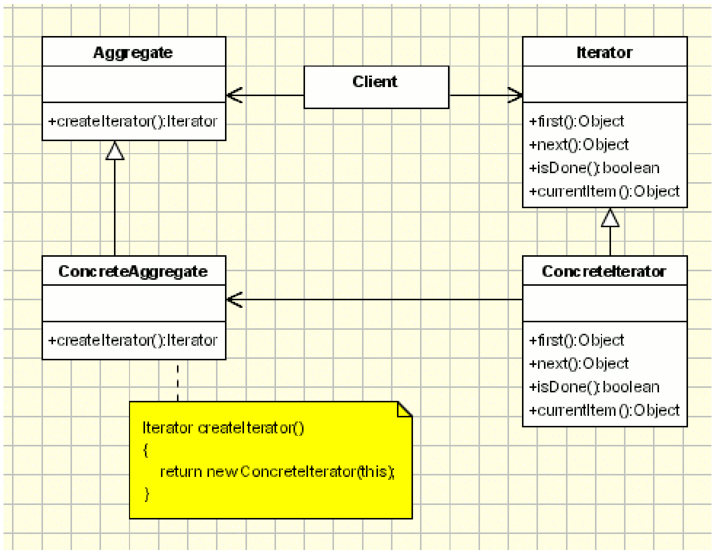
The Iterator Pattern

Design Patterns

Alex Lang

9/1/16

For this assignment, we were told to program the iterator pattern. For this project, we were to find a topic and find a way to iterate it in two separate ways. While there are many different data types that you can use with an iterator, I chose to go with the list, as it fit my topic very well and the video explanation went over this method as well. For my project, I chose to iterate the amount of points the Cleveland Cavaliers scored over the course of their 2015-2016 season by most scored and by least scored.



```
Iterator createIterator()
{
    return new ConcreteIterator(this);
}
```

Pictured to the right is the UML diagram for the Iterator Pattern (diagram from oodesign.com). The diagram displays the absolute minimum necessities to creating the iterator pattern. Aggregate and Iterator are abstract classes, and are inherited by ConcreteAggregate and ConcreteIterator respectively. Everything from those classes are drawn together in the client to be able to iterate from data in whichever sort of way you choose to sort it. For my particular version of the iterator pattern, I actually have a couple of different ConcreteIterator classes, as was needed since I needed to iterate my data in at least two separate ways.

For the iterator class, I established the abstract methods that would be later used by the iterator classes MostConcreteIterator and LeastConcreteIterator. In the iterator class, it is declared that the methods are abstract, the method name and what they are returning, as shown below.

```
    public abstract class Iterator     //This is the iterator class, which is inhereted by
                                                                the Concrete Iterator class
                                        //Does not rely on anything
    {
        public abstract object first();                        //abstract methods being
inherited to MostConcreteIterator and LeastConcreteIterator.
        public abstract object next();
        public abstract bool isDone();
        public abstract object currItem();
    }
```

The reason created the iterator class first before all the other classes is because it was the only class in the diagram that did not rely on any of the other sections. After creating the iterator class, I could now move on to making the aggregate class, since I now had an iterator object I could return.

```
public abstract class Aggregate //Aggregate Class, the Concrete Aggregate Class inherits
this
    {
        public List<string> Elements;      //abstract methods and declaration for the
Concrete Aggregate class to use.

        public abstract Iterator createMostIterator();
        public abstract Iterator createLeastIterator();
    }
```

The Aggregate class, much like the iterator class, is an abstract class. It is inherited by the ConcreteAggregate class. In this Aggregate class, I declared how I am going to be storing our data, which in this case, is strings in a list. The createMostIterator and createLeastIterator are public abstract methods that serve to create both iterators from the aggregate data that is being sent into it.

With both abstract classes out of the way, I created the ConcreteAggregate, MostConcreteIterator and LeastConcreteIterator classes. ConcreteAggregate inherits from the

Aggregate class. The ConcreteAggregate class is the class that is mostly responsible for gathering the data that we enter and getting it ready to iterate. The code below is what I created for this class.

```
public class ConcreteAggregate : Aggregate //ConcreteAggregate Class inherits from
aggregate
    {

        public ConcreteAggregate()
        {
            Elements = new List<string>();            //Establishes where our information
will be stored, in a list as strings.
        }
        public override Iterator createMostIterator()
        {
            return new MostConcreteIterator(this);   //Creates the information for the
most iterator with this aggregate information.
        }

        public override Iterator createLeastIterator() //Creates the information for the
least iterator with this aggregate information.
        {
            return new LeastConcreteIterator(this);
        }
    }
```

ConcreteAggregate( ) servers as a constructor. It creates the elements from the list of strings, and gives the iterators the data it will be using. In createMostIterator(), the method returns a new instance of MostConcreteIterator(this), meaning it returns MostConcreteIterator using this aggregate data. The same goes for the LeastConcreteIterator. The reason that both iterators are in this constructor is because they are using the same data as each other. The only difference between the two is HOW they are iterating it.

The MostConcreteIterator inherits from the iterator class, and defines what each method does. In the most iterator, I programmed it so it was iterate between the items in the list by the most amound of points scoring. So with this iterator, I wanted to start from the bottom of the array and work my way up.

```
public class MostConcreteIterator : Iterator
    {
        Aggregate aggregate;                               //Creates instance for 
aggregate
        int currIndex;

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

        public override object currItem()                  //retrieves the current item 
of the elements list
        {
            if (isDone())                                  //If the iterator is done 
going through the list, return null, else, return the value
                return null;
            return aggregate.Elements[currIndex];
        }

        public override object first()                     //Since this is increasing, 
the index starts at 0.
        {
            currIndex = 0;
            return currItem();
        }

        public override bool isDone()                      //If the index goes the amount 
of elements we have, the iterator is done
        {
            return (currIndex > aggregate.Elements.Count - 1);
        }

        public override object next()                      //If the iterator isn't done, 
go forwards in the array. Return the value.
        {
            if (!isDone())
                currIndex++;
            return currItem();

        }
           }
```

The first() method for this particular iterator establishes that the array starts at 0, and returns the current item at that index. The next() method adds one to the index if the array is not at the end already. To check if the array is at the end, the isDone() method was created, and returns true if the index is greater than the amount of elements we have.

The LeastConcreteIterator is extremely similar to the MostConcreteIterator. The only difference between the two is where they start, where they end, and how next works.

```
public class LeastConcreteIterator : Iterator
    {
        Aggregate aggregate;             //creates instance for aggregate
        int currIndex;

        public LeastConcreteIterator(Aggregate agg)
        {
            aggregate = agg;
        }
        public override object currItem()             //retrieves the current item of the
elements list
        {
            if (isDone())                            //If the iterator is done going
through the list, return null, else, return the value
                return null;
            return aggregate.Elements[currIndex];
        }

        public override object first()                //Since this is decreasing, the
index starts at 14.
        {
            currIndex = 14;
            return currItem();
        }

        public override bool isDone()                //If the index goes below 0, the
iterator is done
        {
            return (currIndex < 0);
        }

        public override object next()                //If the iterator isn't done, go
backwards in the array. Return the value.
        {
            if (!isDone())
                currIndex--;
            return currItem();

        }
           }
```

The first() method for this iterator starts at 14, since the list of the data I am using has 14 elements. Since we are going backwards however, the next() method removes one from the index. The isDone() method is different in this iterator from the other since it checks wether or not the iterator has gone below zero, since that is the lowest number address an array can have.

With all these classes created, putting them all together with the form was the easy part. Before any other coding, I made sure to a new aggregate instance from the concreteAggregate constructor, so the data can be loaded into the list properly. The other iterators, MostConcreteIterator and LeastConcreteIterator are declared there as well. Another method was created for the form constructor, so that the information could be loaded into the aggregate class to get ready for the iterator.

```
private void prepareAggWithIter()
        {
            agg.Elements.Add("Lebron James              1920");
            agg.Elements.Add("Kevin Love                  1234");
            agg.Elements.Add("Kyrie Irving                   1041");
            agg.Elements.Add("J.R. Smith                      955");
            agg.Elements.Add("Tristan Thompson         643");
            agg.Elements.Add("Matthew Dellavedova   569");
            agg.Elements.Add("Timofey Mozgov            475");
            agg.Elements.Add("Channing Frye               425");
            agg.Elements.Add("Richard Jefferson          410");
            agg.Elements.Add("Mo Williams                   338");
            agg.Elements.Add("Iman Shumpert              311");
            agg.Elements.Add("James Jones                 178");
            agg.Elements.Add("Jordan McRae                99");
            agg.Elements.Add("Sasha Kaun                    23");
            agg.Elements.Add("Dahntay Jones                13");
            mostIterator = agg.createMostIterator();
            leastIterator = agg.createLeastIterator();
               }
```

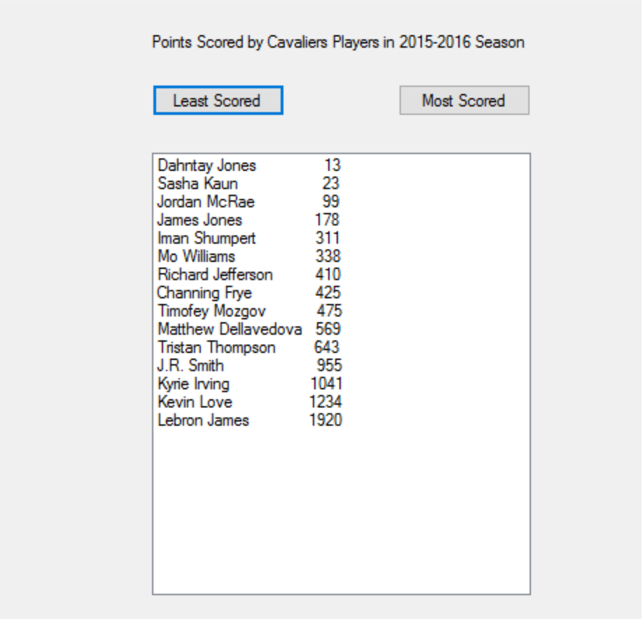
It is in this method where mostIterator and leastIterator are actually defined, and are created from the concreteAggregate Class. With everything loaded in, all I had left to do with iterate this information.

My app had two separate buttons, one to sort from most amount of points, and one to sort from least amount of points.

```
private void iterateLeastBtn_Click(object sender, EventArgs e)
        {
            iterateList.Items.Clear();

            for (leastIterator.first(); !leastIterator.isDone(); leastIterator.next())
            {
                iterateList.Items.Add(leastIterator.currItem());
            }
        }

        private void iterateMostButton_Click(object sender, EventArgs e)
        {
            iterateList.Items.Clear();

            for (mostIterator.first(); !mostIterator.isDone(); mostIterator.next())
            {
                iterateList.Items.Add(mostIterator.currItem());
            }

               }
```

Both buttons add the elements from their lists to the list box in their own particular order how they were arranged. To look nicer and not have extremely long lists, I also made sure to clear any existing list already there with the Clear() method.

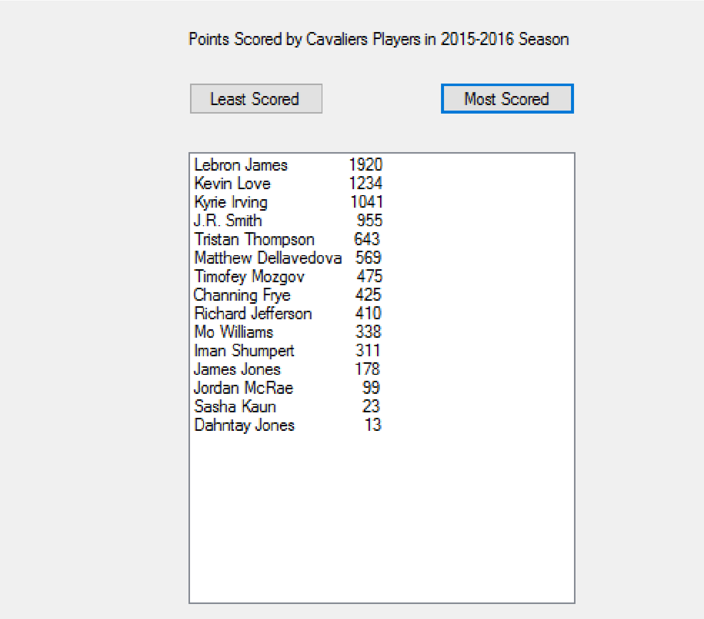
When sorted by least, the players appear in the order of who scored the fewest points in the season.



Points Scored by Cavaliers Players in 2015-2016 Season

```
Dahntay Jones         13
Sasha Kaun            23
Jordan McRae          99
James Jones          178
Iman Shumpert        311
Mo Williams          338
Richard Jefferson    410
Channing Frye        425
Timofey Mozgov       475
Matthew Dellavedova  569
Tristan Thompson     643
J.R. Smith           955
Kyrie Irving        1041
Kevin Love          1234
Lebron James        1920
```

When sorted by most, the players appear in the order of who scored the most amount of points in the season.



Points Scored by Cavaliers Players in 2015-2016 Season

```
Lebron James         1920
Kevin Love           1234
Kyrie Irving         1041
J.R. Smith            955
Tristan Thompson      643
Matthew Dellavedova   569
Timofey Mozgov        475
Channing Frye         425
Richard Jefferson     410
Mo Williams           338
Iman Shumpert         311
James Jones           178
Jordan McRae           99
Sasha Kaun             23
Dahntay Jones          13
```

Conclusion: This project started off very confusing for me. The first day when you were teaching us about all these design patterns and showing us the UML Chart I was really blown back. The week of you teaching this helped me understand this so much more than I ever thought I would. I honestly think the reason why I was able to do this so well was because of the notes I took in class, and tried very hard to pay as much attention as I can. The video was also extremely helpful, and even in the middle of the video when I sort of fell behind a bit, I appreciated how you went back and walked the viewers through to make sure we understood what we did so far. This project was challenging, not too difficult, but definitely required a lot of work, and it was a solid design pattern to start off with.