The Iterator Pattern

Design Patterns

Mark Swarner

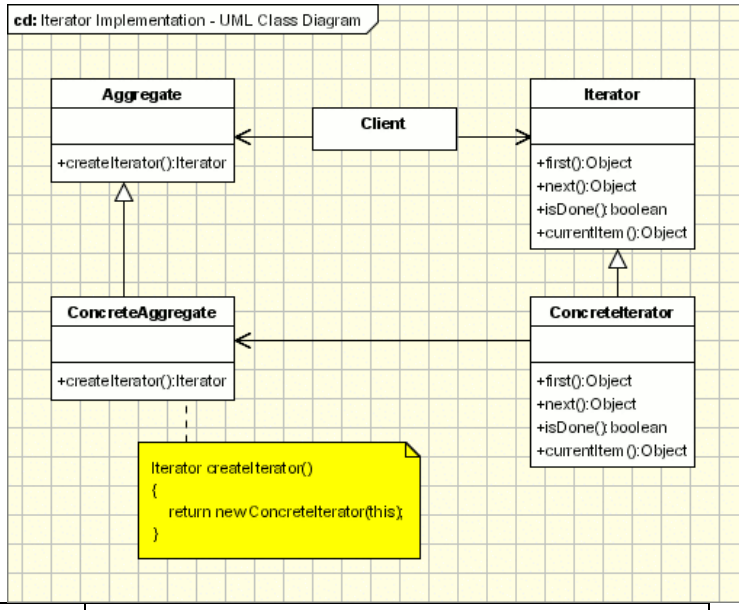
Fall 2016-17

Introduction:

This is an assignment where we were required to write a program in C# that correctly implements the iterator pattern. The following submission iterates through two List<String> data structures.

The UML Diagram For The Iterator Pattern:

The UML Diagram shown to the right illustrates the basic structure of a program utilizing the iterator pattern. It also describes the individual classes and functions. The table below explains how I utilized the structure:



Aggregate	I created an abstract class named Aggregate to fill this role.
Iterator	I created an abstract class named Iterator to fill this role.
ConcreteAggregate	I created a class named ConcreteAggregate, which derives from aggregate, to fill this role.
ConcreteIterator	I created a class named ConcreteIterator, which derives from Iterator, to fill this role.
Client	The demo application fulfills this role.

Narrative:

```
public abstract class Aggregate
    {
        public abstract Iterator
createIterator();
    }
```

The Aggregate class has the modifier abstract. This means that all I can do is create function prototypes, but it also allows for me to derive from this class.

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

The Iterator class works in the same way as the Aggregate class. However, I did have to add arguments to the functions to be able to setup multiple iterators. These arguments will be explained later.

```
class ConcreteAggregate :
Aggregate
    {
        public List<String> bands
= new List<String>();
        public List<String> albums
= new List<String>();

        public void createData()
        {
        }

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

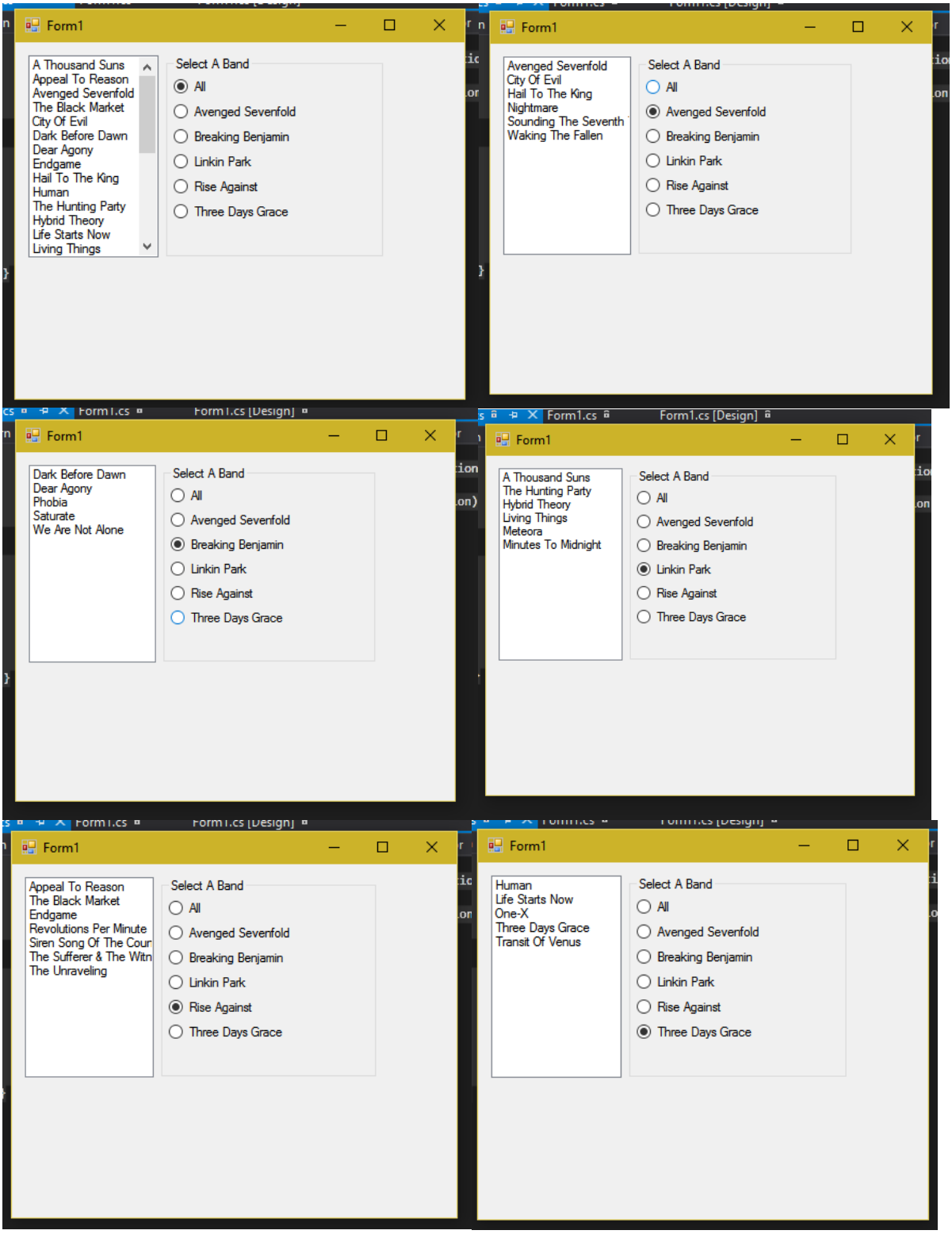
This is the class ConcreteAggregate, which derives from Aggregate. This class creates the data structures we need as well as creates an iterator. I left the code from the createData() function out due to lack of space but all that it does is add the data I specified to each of the two lists.

```
class ConcreteIterator : Iterator
    {
        public int
currentItemPosition;
        ConcreteAggregate
aggregate;
        public
ConcreteIterator(ConcreteAggregate
cAggregate)
        {
            aggregate =
cAggregate;
        }
        public override String
first(String listName)
        {
            currentItemPosition =
0;
            return
currentItem(listName);
        }

        public override String
next(String listName)
        {
            currentItemPosition++;
            return
currentItem(listName);
        }

        public override bool
isDone(String listName)
        {
            if(listName ==
"Albums")
                return
(aggregate.albums.Count - 1 ==
currentItemPosition);
            else
                return
(aggregate.bands.Count - 1 ==
currentItemPosition);
        }

        public override String
currentItem(String listName)
        {
            if (listName ==
"Albums")
                return
aggregate.albums[currentItemPositi
on];
            else
                return
aggregate.bands[currentItemPositio
n];
        }
    }
```

This is the ConcreteIterator class. The meat and bones are contained within this class as none of the operations necessary to have an iterator class can happen without the function definitions contained within this class. listName is used as a way of determining which list to operate on when a function is called.

```
private void
riseAgainst_rbtn_CheckedChanged(
object sender, EventArgs e)
        {
            if
(riseAgainst_rbtn.Checked ==
true)
            {

cIterator.first("Bands");

                while
(!cIterator.isDone("Bands"))
                {
                    if
(cIterator.currentItem("Bands")
== "Rise Against")
                    {

lbDisplay.Items.Add(cIterator.cu
rrentItem("Albums"));

cIterator.next("Bands");
                    }
                    else

cIterator.next("Bands");
                }
            }
            else
            {

lbDisplay.Items.Clear();
            }
        }
```

This is an example of the code for the band radio buttons. The program is constantly checking for a change in state (From checked to unchecked or unchecked to checked) for each button and whenever there I a change, it implements one of two sections. If the button becomes selected, then it implements the code within the if statement. If it becomes unselected, it erases all data from the listbox in order to make room for the new data.


Form1
A Thousand Suns
Appeal To Reason
Avenged Sevenfold
The Black Market
City Of Evil
Dark Before Dawn
Dear Agony
Endgame
Hail To The King
Human
The Hunting Party
Hybrid Theory
Life Starts Now
Living Things
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace
Form1
Avenged Sevenfold
City Of Evil
Hail To The King
Nightmare
Sounding The Seventh
Waking The Fallen
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace
Form1.cs
Form1.cs [Design]
Form1
Dark Before Dawn
Dear Agony
Phobia
Saturate
We Are Not Alone
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace
Form1.cs
Form1.cs [Design]
Form1
A Thousand Suns
The Hunting Party
Hybrid Theory
Living Things
Meteora
Minutes To Midnight
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace
Form1.cs
Form1.cs [Design]
Form1
Appeal To Reason
The Black Market
Endgame
Revolutions Per Minute
Siren Song Of The Cour
The Sufferer & The Witn
The Unraveling
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace
Form1
Human
Life Starts Now
One-X
Three Days Grace
Transit Of Venus
Select A Band
All
Avenged Sevenfold
Breaking Benjamin
Linkin Park
Rise Against
Three Days Grace

The screenshots above show each radio buttons results.

Observations:

Overall I mostly enjoyed this. I learned a few things that will actually help me on personal projects in the near future. The iterator pattern seems scary at first but is relatively nice and has an abundance of uses.