38. Iterator

Iterator is a design pattern that allows a developer to encapsulate a collection of any complexity inside an object with a very simple interface. The consumers of this interface would be able to iterate through each individual item of this collection without knowing any of its implementation details.

In C#, pretty much any inbuilt collection data types that can be traversed inside a `for each` loop are based on the Iterator design pattern. For example, they expose methods such as `MoveNext`, which are commonly used by this design pattern.

Iterator design pattern can be summarized as follows:

- There is an object known either as an **Aggregate** or a **Collection**. This object stores the actual collection of objects. This collection can be of any structure and any complexity (trees, stacks, arrays, etc.).
- There is an object called **Iterator**, the role of which is to read the items from the **Aggregate** and expose them to the outside world one by one.
- **Iterator** object reads items in a specific order and keeps track of the current item.

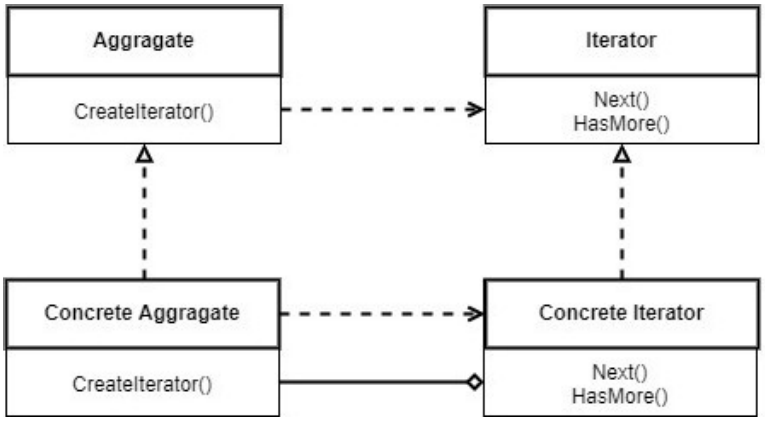

Figure 38.1 - Iterator UML diagram

We will now go through an example implementation of Iterator. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Iterator

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Iterator implementation example

We will create a console application project. And the first thing that we will add to this project would be the following interface:

```
namespace Iterator_Demo;

internal interface IIterator
{
    bool MoveNext();
    int GetCurrent();
}
```

This is our **Iterator** interface. It has two methods: `MoveNext` and `GetCurrent`. The `MoveNext` method would iterate through each item of the collection and update the pointer to the current item. It will return `true` if it is possible to move to the next item and it will return `false` if there are no more items left.

`GetCurrent` will retrieve the current value from the collection. For the sake of simplicity, we are dealing with the collections of `int` in our example. However, in a real-life application, you may want your **Itrator** to be generic and get it to work with collections of any data type.

Next, we will create an interface for our **Aggregate**, which will be as follows:

```
namespace Iterator_Demo;

internal interface IAggregate
{
        IIterator CreateIterator();
        void Insert(int value);
}
```

In our case, this interface allows us to retrieve the **Iterator** and insert an item into the collection.

Next, we will add a concrete **Aggregate** that implements this interface. We will start with a basic **Aggregate** that simply encapsulates a `List` of values.

```
namespace Iterator_Demo;

internal class ListAggregate : IAggregate
{
        private readonly List<int> collection;

        public ListAggregate()
    {
        collection = [];
    }

    public IIterator CreateIterator()
    {
        return new ListIterator(this);
    }

    // Get item count
    public int Count
    {
        get { return collection.Count; }
    }

    // Indexer
    public int this[int index]
    {
        get { return collection[index]; }
        set { collection.Insert(index, value); }
    }

    public void Insert(int value)
    {
        collection.Add(value);
    }
}
```

Then, we will add an **Iterator** for this **Aggregate** type. All we do

inside this iterator is go through each item of the collection, which maintains the index of the current item. If there are no more items left, the `MoveNext` method will return `false`.

```
namespace Iterator_Demo;

internal class ListIterator(
    ListAggregate aggregate) : IIterator
{
        private int currentIndex = -1;

    public bool MoveNext()
        {
                if (currentIndex + 1 < aggregate.Count)
                {
                        currentIndex++;
                        return true;
                }

                return false;
        }

        public int GetCurrent()
        {
                return aggregate[currentIndex];

        }
}
```

But now, we will add something more interesting. What if we had an **Aggregate** that stores data on a sorted binary tree? This would be more complicated than dealing with a flat `List`. And this is where the Iterator design pattern truly shows its usefulness.

Before we add the new **Aggregate**, we will need to add an object that will represent a node on such a tree. And this is what it will

look like:

```
namespace Iterator_Demo;

internal class Node(int value)
{
    public int Value { get; set; } = value;
    public Node? Left { get; set; }
        public Node? Right { get; set; }
        public Node? Parent { get; set; }
}
```

We can now start adding our **Aggregate** that will deal with such a data type. We have called our class `SortedBinaryTreeCollection` to demonstrate that **Aggregate** and **Collection** are interchangeable names in the context of the Iterator design pattern.

```
namespace Iterator_Demo;

internal class SortedBinaryTreeCollection : IAggregate
{
    private Node? root;

    public SortedBinaryTreeCollection()
    {
        root = null;
    }

    public IIterator CreateIterator()
    {
        return new SortedBinaryTreeIterator(this);
    }

    public Node? GetFirst()
    {
```

```
        var current = root;

        while (true)
        {
            if (current?.Left is not null)
            {
                current = current.Left;
            }
            else
            {
                return current;
            }
        }
    }
}
```

Then, we will add the following method, which will allow us to insert new values into the tree, while maintaining all the values in the right order:

```
public void Insert(int value)
{
    Node newNode = new(value);

    if (root is null)
    {
        root = newNode;
    }
    else
    {
        Node parent;
        var temp = root;

        while (true)
        {
```

```
16              parent = temp;
17
18              if (value < temp.Value)
19              {
20                  temp = temp.Left;
21
22                  if (temp is null)
23                  {
24                      parent.Left =
25                          newNode;
26                      newNode.Parent =
27                          parent;
28                      return;
29                  }
30              }
31              else
32              {
33                  temp = temp.Right;
34
35                  if (temp is null)
36                  {
37                      parent.Right =
38                          newNode;
39                      newNode.Parent =
40                          parent;
41                      return;
42                  }
43              }
44          }
45      }
46  }
```

Now, we can add the **Iterator** object for our sorted binary tree collection:

```
amespace Iterator_Demo;

internal class SortedBinaryTreeIterator(
    SortedBinaryTreeCollection aggregate) :
    IIterator
{
        private Node? current = null;

    public int GetCurrent()
    {
        return current?.Value ?? 0;
    }
}
```

We will then add the following public method to it to allow us to traverse the tree from left to right:

```
public bool MoveNext()
{
    if (current is null)
    {
        current = aggregate.GetFirst();
        return true;
    }

    if (current.Right is not null)
    {
        current = current.Right;

        while (true)
        {
            if (current.Left is null)
            {
                break;
            }
```

```
19
20              current = current.Left;
21          }
22
23          return true;
24      }
25      else
26      {
27          var originalValue = current.Value;
28
29          while (true)
30          {
31              if (current.Parent is not null)
32              {
33                  current = current.Parent;
34
35                  if (current.Value > originalValue)
36                  {
37                      return true;
38                  }
39              }
40              else
41              {
42                  return false;
43              }
44          }
45      }
46  }
```

So, the consumer of this **Iterator** doesn't have to know anything about the tree structure. All it cares about is that, while there are still items in the tree, it can retrieve them all one by one.

Now, we can examine the differences in the output of our two **Iterator** objects. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Iterator_Demo;

var listOfValues = [
    8, 19, 25, 2, 4, 7, 32, 90, 3, 1
];

var listAggregate =new ListAggregate();

foreach (var value in listOfValues)
{
    listAggregate.Insert(value);
}

var listIterator = listAggregate
    .CreateIterator();

Console.WriteLine(
    "Values from List Iterator:");

while (listIterator.MoveNext())
{
    Console.Write(
        $"{listIterator.GetCurrent()}, ");
}

var treeAggregate =
    new SortedBinaryTreeCollection();

foreach (var value in listOfValues)
{
    treeAggregate.Insert(value);
}

var treeIterator = treeAggregate
    .CreateIterator();
```

```
36
37  Console.WriteLine(string.Empty);
38  Console.WriteLine(
39      "Values from Binary Tree Iterator:");
40
41  while (treeIterator.MoveNext())
42  {
43      Console.Write(
44          $"{treeIterator.GetCurrent()}, ");
45  }
46
47  Console.ReadKey();
```

And, as we can see in the screenshot below, even though we have used identical inputs for both of the collection types, the collection inside `ListAggregate` was identical to the input collection, while the collection inside `SortedBinaryTreeCollection` was returned sorted. And this is because the former **Aggregate** used just a plain `List`, while the latter used a sorted binary tree, which was traversed from left to right.

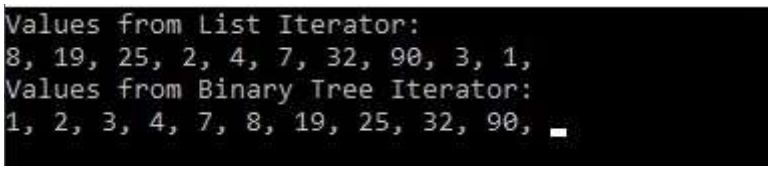

Figure 38.2 - Two iterators showing different results

This concludes our example of the Iterator design pattern. Let's now summarize all its key benefits.

Benefits of using Iterator

The main benefits of using Iterator design pattern are as follows:

- Single responsibility principle is well maintained, as maintaining the collection and iterating through it are performed by different objects.
- If you are dealing with complex data structures, the Iterator design pattern significantly simplifies this process.
- Because you can create more than one **Iterator** from each **Aggregate**, you can iterate through the same collection in parallel.

And now let's examine some things that you should watch out for while using the Iterator pattern.

Caveats of using Iterator

If we go back to our `ListAggregate`, you will see that we have hardly added any value to it. The object merely encapsulates an inbuilt `List` data type. This data type has its own **Iterator**, which, in the language of the C# system library is called Enumerator.

This demonstrates the first problem with using the Iterator design pattern. If you are dealing with simple collections, it would be better to just deal with them directly. If you apply the Iterator design pattern in such a situation, you will only make your code more complicated without adding any value at all. So, you should only reserve this design pattern for complex collection types.

But even while dealing with complex collection types, it is worth checking whether it wouldn't be more efficient to just traverse the collection directly. Even though the Iterator design pattern allows you to provide a convenient interface that enables the consuming client to traverse the collection in one direction, it could be that for certain collection types traversing it in one direction wouldn't be the most appropriate or the most efficient thing to do.