M S RAMAIAH INSTITUTE OF TECHNOLOGY

(Autonomous Institute Affiliated to VTU)

Department of Information Science and Engineering

Assignment Report

USN & Name:	1MS17IS077 – Nimish Nikhil Bongale
Design Pattern:	Iterator
Course Code and Name:	IS62B Object Oriented Analysis and Design Patterns

ITERATOR PATTERN

Gang of Four Definition

"Provide a way to access the elements of an aggregate object one after the other without exposing its underlying representation."

Key Features

Intent

Simplify the process of iterating through objects by creating a common universal interface.

An aggregate object contains other objects for the purpose of grouping those objects as a unit (Wrapping/Encapsulation). It is also called a container or a collection. Some commonly known examples are linked list and a hash table.

Problem

To Iterate through a certain list of items without interfering with the inner working of the items.

Motivation

An aggregate object such as a list should, in essence, allow a way to traverse its elements without exposing its internal structure. It may also allow different traversal methods. It could also allow multiple traversals to be in progress at the same time. But, ideally, we really do not want to add all these methods to the interface for the aggregate.

Solution

An iterator designed using the iterator design pattern interface.

Participants

1. Iterator
   › Defines the basic interface for accessing and traversing elements

2. ConcreteIterator
   › Implements the "Iterator" interface
   › Keeps track of the current position *i* in the traversal from *x*->*y*, where *x*<=*i*<=*y*.

3. Aggregate
   › Defines an interface for creating an Iterator object (which is internally a factory method)

4. ConcreteAggregate
   › Implements the Iterator creation interface to also return an instance of the proper ConcreteIterator.

Collaborations
A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

Pros

› Simplifies the interface of the Aggregate (as the methods are not implemented within it)
› Supports multiple concurrent traversals
› Supports variant traversal techniques

Cons

› Implementation of an Iterator may be tricky.
› Dealing with change in the underlying data structures is difficult.
› Must support an additional hasNext() method to check if there is an element existing after the current one.

Implementation

The iteration is controlled by either the client or by the iterator itself. If an external client is involved, it is known as an external iterator; otherwise internal. The traversal algorithm is defined by the iterator or sometimes the aggregate. Common implementations have it defined in the iterator, as it becomes easier to have variant traversal techniques. An iterator that allows insertion and deletions without affecting the traversal is called a robust iterator.

Case Study

An SOS/Hotkeys messaging service

To understand the iterator pattern, we can take the case of an SOS messaging service. To message an SOS number, there is sometimes a specific hotkey pattern or a set sequence which when typed in to the number pad will message the numbers previously stored. When this sequence is punched in, we will verify it, iterate through the stored numbers, and message each person.

Let’s take this understanding of this example to the next level. Suppose we are mimicking MessageApp in the users phone that sends an automated message and a confirmation of the same arrives back in the app after the SOS message is delievered. SOSCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (MessageApp).

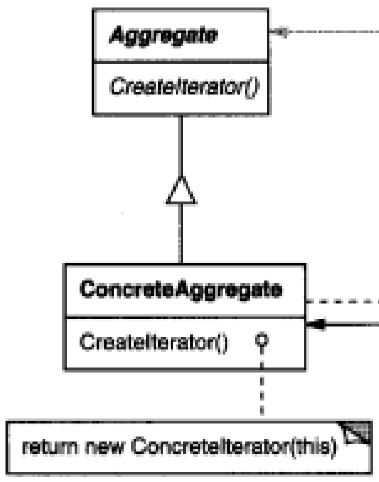
Our Main (Driver) class instantiates an SOScollection which by default adds a few phone numbers to a list of SOS Phone numbers. A max of 3 phone numbers may be added to the SOS Hotkey list. The SOSCollection class implements the Collection Interface. The Collection interface contains the createIterator() method which returns an iterator after its created.

Our SOSIterator class implements Iterator, which has 2 basic functions, next() and hasNext(). next() returns the current item in the iteration and increments the position, whereas hasNext() checks if there is an item after the current one.

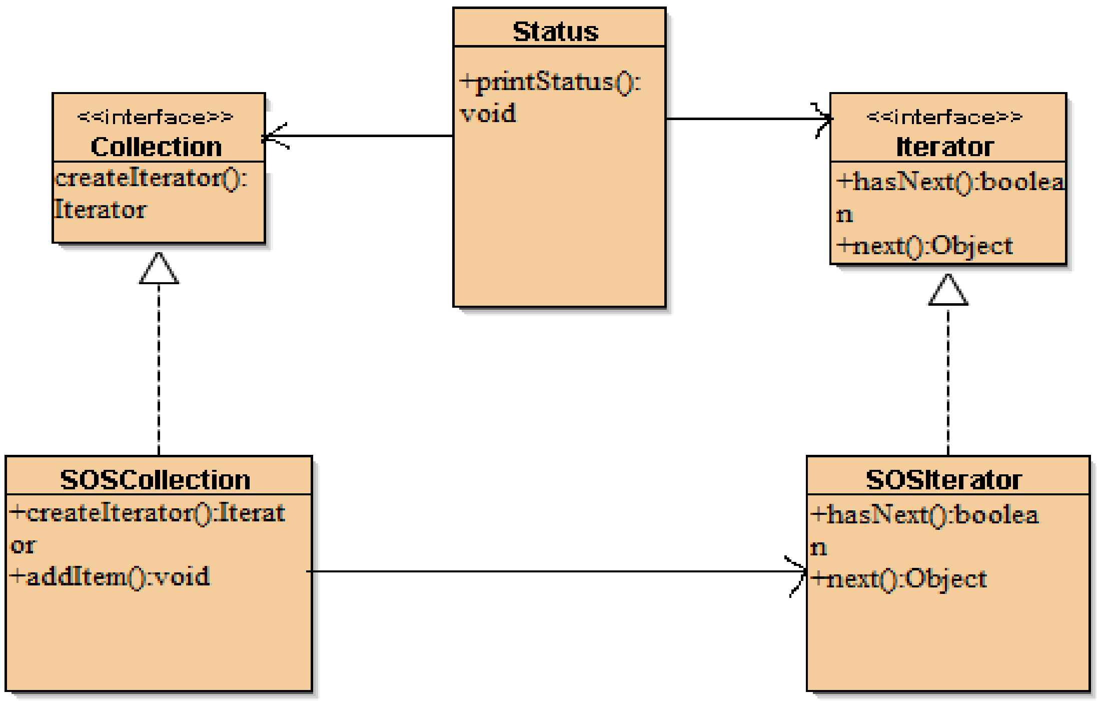
Our class MessageApp is essentially the client. Its object is instantiated with a list of SOS contacts which it feeds into an SOSCollection object. It then proceeds to create an Iterator over this SOSCollection object. It also contains a method called as printStatus() which prints the status of the app at any given point of time. (Ready, Sending, Delivered). There is a certain time delay in between these states to mimick the sending and receiving of messages.

As a result of this implementation, a crucial decoupling is achieved as the iterator is able to iterate through any kinds of objects. Even if we would have used ArrayList instead of Array, or even LinkedList, or any generic class template there will not be any change in the client (Status) code due to the decoupling achieved by the use of iterator interface.

Generic Class Diagram of Iterator



Case Study Class Diagram



Java Code

SOS.java

```
class SOS

{

    String number;

    String message;

    public SOS(String number)

    {

        this.number = number;

        this.message = "HELP";

    }

    public String getNumber()

    {

        return number;

    }

    public String getMessage()

    {

        return message;

    }

}
```

Collection.java

```
interface Collection

{

    public Iterator createIterator();
}
```

SOSCollection.java

```
class SOSCollection implements Collection
{
    static final int MAX_ITEMS = 3;
    int numberOfItems = 0;
    SOS[] sosList;

    public SOSCollection()
    {
        sosList = new SOS[MAX_ITEMS];
        addItem("9609876543");
        addItem("9087654321");
        addItem("9807654321");
    }

    public void addItem(String str)
    {
        SOS notif = new SOS(str);
        if (numberOfItems >= MAX_ITEMS)
            System.err.println("Full");
        else
        {
            sosList[numberOfItems] = notif;
            numberOfItems = numberOfItems + 1;
        }
    }

    public Iterator createIterator()
    {
        return new SOSIterator(sosList);
    }
}
```

Iterator.java

```
interface Iterator
{
    boolean hasNext();
    Object next();
}
```

SOSIterator.java

```
class SOSIterator implements Iterator
{
    SOS[] sosList;
    int pos = 0;
    public SOSIterator (SOS[] sosList)
    {
        this.sosList = sosList;
    }
    public Object next()
    {
        SOS notif = sosList[pos];
        pos += 1;
        return notif;
    }

    public boolean hasNext()
    {
        if (pos >= sosList.length ||
            sosList[pos] == null)
            return false;
        else
            return true;
    }
}
```

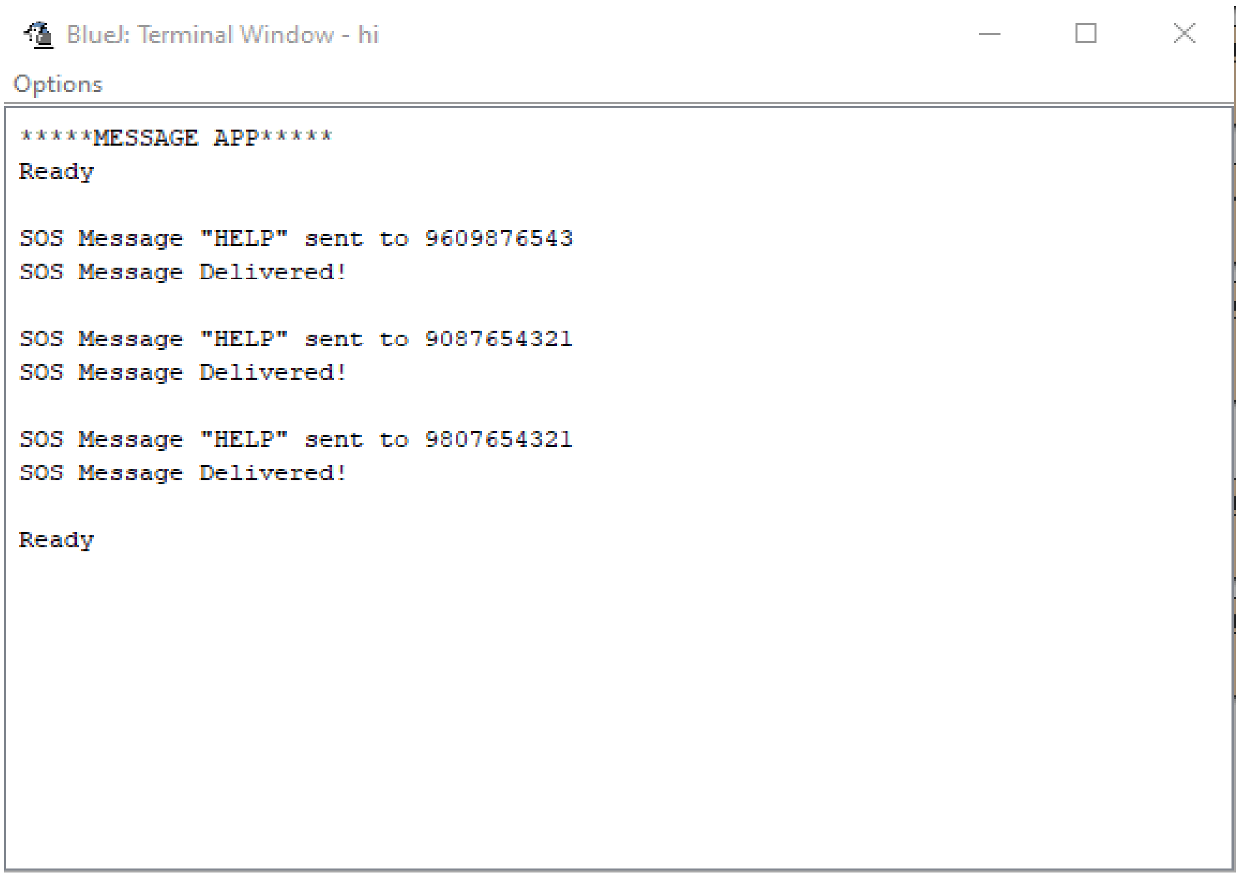
MessageApp.java

```
class MessageApp
{
    SOSCollection notifs;
    public MessageApp(SOSCollection notifs)
    {
        this.notifs = notifs;
    }


    public void printStatus()

    {

        Iterator iterator = notifs.createIterator();

        System.out.println("*****MESSAGE APP*****");

        System.out.println("Ready");

        while (iterator.hasNext())

        {

            SOS n = (SOS)iterator.next();

            System.out.println("\nSOS Message 
"+"\""+n.getMessage()+"\""+" sent to "+n.getNumber());

            try {

            Thread.sleep(1000);

            System.out.println("SOS Message Delivered!");

             Thread.sleep(500);

        }

            catch(Exception e){

            e.printStackTrace();

            }

        }

        System.out.println("\nReady");

    }

}
```

Main.java

```
public class Main

{

    public static void main(String args[])

    {

        SOSCollection nc = new SOSCollection();

        MessageApp ma = new MessageApp(nc);

        ma.printStatus();

    }

}
```

Output



```
*****MESSAGE APP*****
Ready

SOS Message "HELP" sent to 9609876543
SOS Message Delivered!

SOS Message "HELP" sent to 9087654321
SOS Message Delivered!

SOS Message "HELP" sent to 9807654321
SOS Message Delivered!

Ready
```

```
                    *****MESSAGE APP*****

Ready

SOS Message "HELP" sent to 9609876543

SOS Message Delivered!

SOS Message "HELP" sent to 9087654321

SOS Message Delivered!

SOS Message "HELP" sent to 9807654321

SOS Message Delivered

Ready
```