CS445_FALL2017

DEMOSTRATE ITERATOR, OBSERVER AND TEMPLATE DESIGN PATTERN

SAI AYSHWARYALAKSHMI RAVICHANDRAN

A20378306

Iterator Design Pattern

What is Iterator Design Pattern?

The iterator design pattern hides the actual implementation of traversal through the collection and the client programs just use the iterator methods for this purpose.

This can be seen in JAVA as Collection framework. The iterator design pattern is aimed at providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The abstraction provided by the iterator design pattern allows you to modify the collection implementation without making any changes outside of the collection.

It enables you to create a general-purpose GUI component that will be able to iterate through any collection of the application.

Why Do we need Iterator Design Pattern?

The Need to access the elements in a collection without exposing its internal representation.

Also, the need to access the list in different ways depending on what needs to be accomplished. The iterator design pattern has an iterator interface which provides a simple interaction mechanism.

How Iterator Design Pattern Works?

Structure of the Iterator Design Pattern

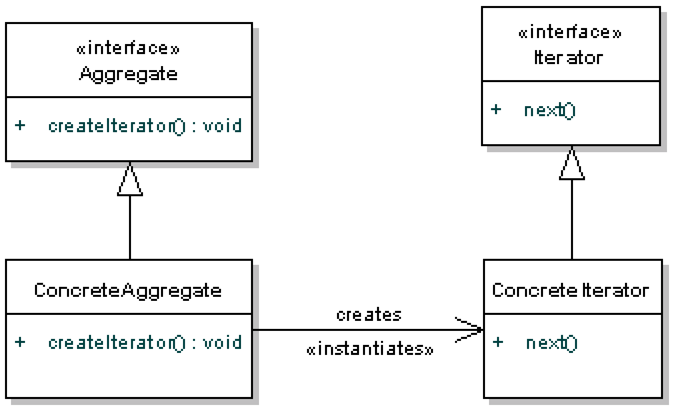


Fig 1 [1]

The Aggregate defines an interface for the creation of the Iterator object.

The Concrete Aggregate implements this interface, and returns an instance of the Concrete Iterator.

The Iterator defines the interface for access and traversal of the elements.

Concrete Iterator implements this interface while keeping track of the current position in the traversal of the Aggregate.

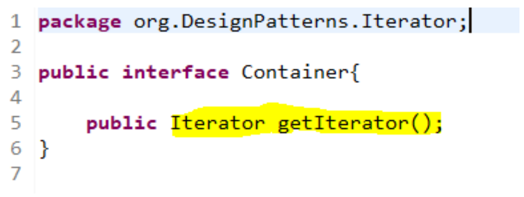
Let us look at an example of a List of Cars.

The Aggregate or Container here will create the new Iterator.

The iterator interface will implement the traversal functions for elements such as next() and hasNext().

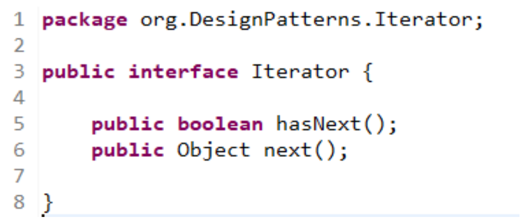
In this example the concrete iterator is Concrete Iterator class and the Concrete Aggregate is CarsList Class.

Let us look at the code:



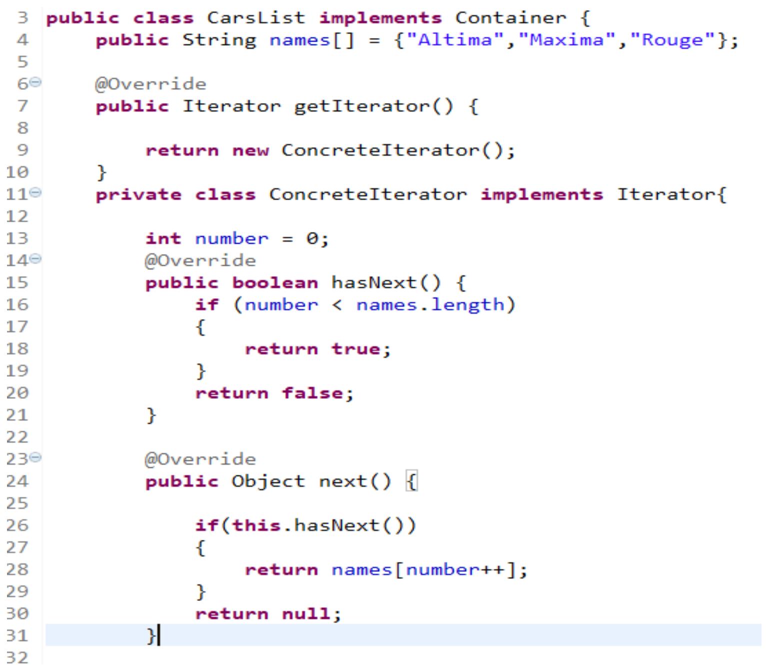
```
1 package org.DesignPatterns.Iterator;
2
3 public interface Container{
4
5     public Iterator getIterator();
6 }
7
```

In the highlighted portion on line 5 in the above code, we can see that the getIterator() method which creates a new Iterator is of Iterator type. This Iterator type is referring to the Iterator interface that is written later in the program.



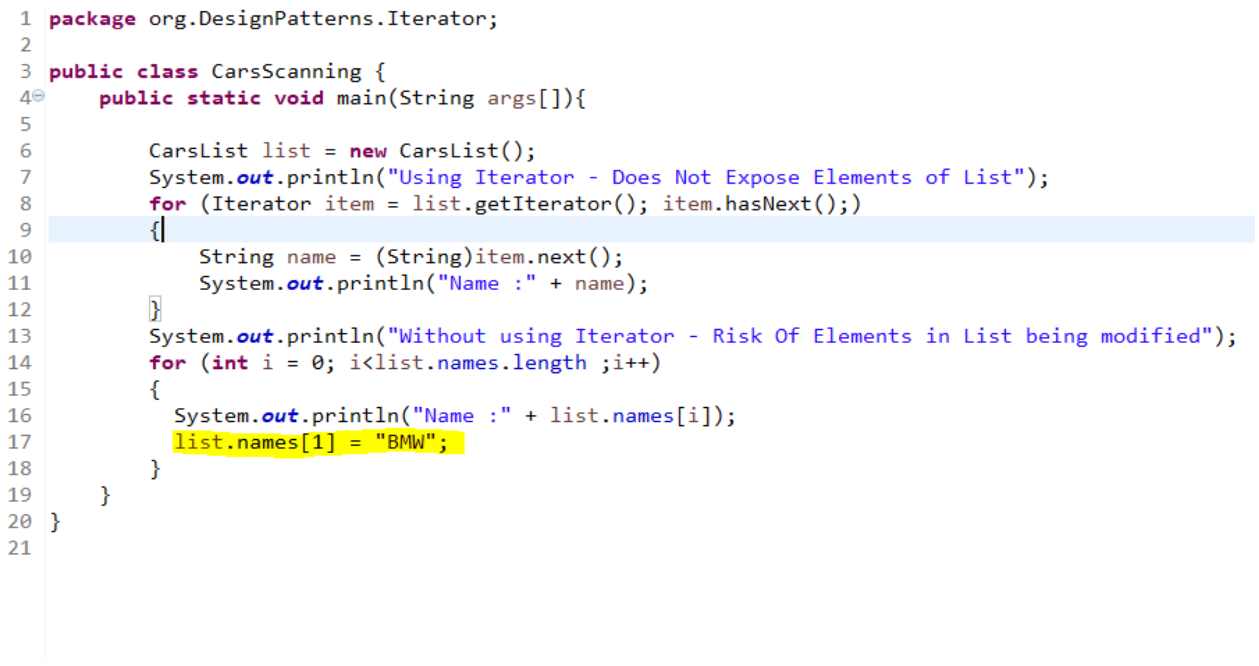
```
package org.DesignPatterns.Iterator;

public interface Iterator {

    public boolean hasNext();
    public Object next();

}
```

When a concrete Aggregate or Collection and a Concrete Iterator implements the above interfaces to iterate through a list of cars , the code can be thought of as follows.



```
public class CarsList implements Container {
    public String names[] = {"Altima","Maxima","Rouge"};

    @Override
    public Iterator getIterator() {

        return new ConcreteIterator();
    }
    private class ConcreteIterator implements Iterator{

        int number = 0;
        @Override
        public boolean hasNext() {
            if (number < names.length)
            {
                return true;
            }
            return false;
        }

        @Override
        public Object next() {

            if(this.hasNext())
            {
                return names[number++];
            }
            return null;
        }

```

Once all the code is implemented, we can try accessing these iterator methods that have just been created.

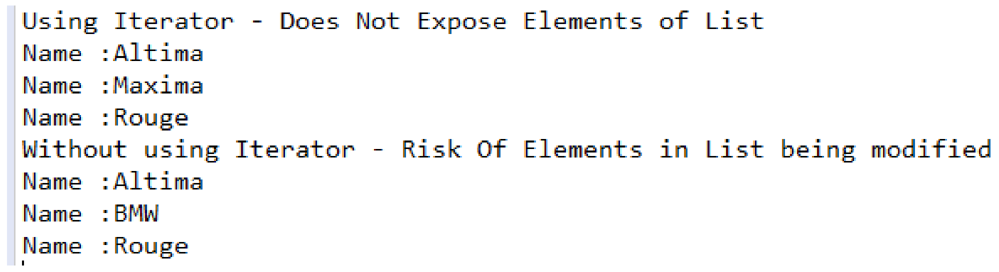


```
package org.DesignPatterns.Iterator;

public class CarsScanning {
    public static void main(String args[]){

        CarsList list = new CarsList();
        System.out.println("Using Iterator - Does Not Expose Elements of List");
        for (Iterator item = list.getIterator(); item.hasNext();)
        {
            String name = (String)item.next();
            System.out.println("Name :" + name);
        }
        System.out.println("Without using Iterator - Risk Of Elements in List being modified");
        for (int i = 0; i<list.names.length ;i++)
        {
          System.out.println("Name :" + list.names[i]);
          list.names[1] = "BMW";
        }
    }
}

```

In the above code we access the elements of the list using the Iterator methods that we have implemented, that provide a level of abstraction for the internal representation of the list. The underlying list cannot be modified if the client has access only to traverse the list.

On the other hand, A naïve implementation without using the iterator design pattern would be to access the list of cars using the index of the elements. This poses a risk of modifying the elements in the list.

It is clearer from the results below that these two pieces of code yield:



```
Using Iterator - Does Not Expose Elements of List
Name :Altima
Name :Maxima
Name :Rouge
Without using Iterator - Risk Of Elements in List being modified
Name :Altima
Name :BMW
Name :Rouge
```

There are different types of Iterator that can be implemented some of them are polymorphic Iterator, External Iterator, Internal Iterator, Robust Iterator, Null Iterator.

Polymorphic Iterator: The polymorphic Iterator has a common contract across multiple collection. The client program that interacts with the iterator need not be changed.

External Iterator: The iteration is controlled by the collection object or the aggregate object. In this case the client code should explicitly invoke the method to move the pointer to the next element.

Internal Iterator: The iterator controls the iteration. In the internal iterator when an element is accessed, the access pointer will automatically move to the next index by the iterator.

Robust Iterator: It is an iterator that works intact even if the aggregate object is modified. This means that even if an element is added to a list or removed from the list or if the position of the elements are interchanged, the iterator still works as it was intended to.

Null iterator: A tree traversal will return the next element when we travel through the nodes but when we travel through the leaf node and ask for the next element, a null iterator will be returned by the aggregate object indicating that it is a leaf node.

Observer Design Pattern

What is Observer Design Pattern?

It defines a one to many dependency between objects so that when one object changes state, all of it's dependents are notified and updated automatically.

The observer pattern falls under behavioral pattern category. Observer pattern is very common and it is a successful standard for decoupling.

The observer pattern is also known as behavioral pattern, as it's used to form relationships between objects at runtime.

Why do we need Observer Pattern?

When the state of Objects is altered, other objects which are dependent on these objects need to be informed.

To have a good design, is to decouple as much as possible and to reduce dependencies.

The concept of listeners is based on this pattern.

How the Observer Design Pattern Works?

Structure of the Observer Pattern

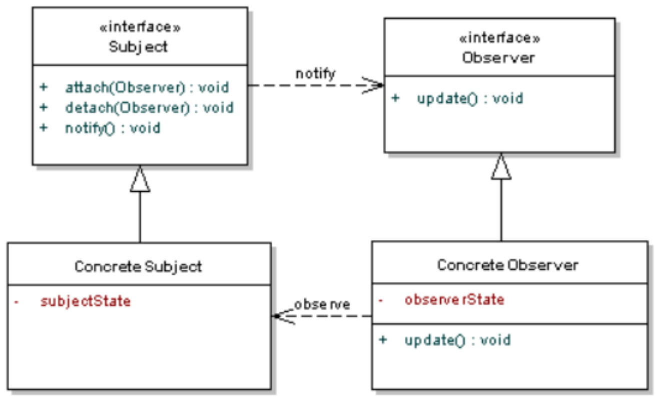
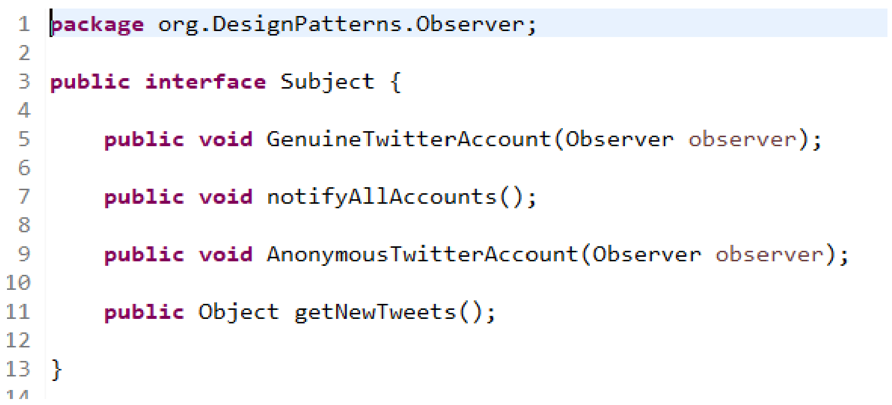


Fig 2 [11]

In this structure there is an Observer which is interested in the state of the subjects and it registers its interest in the Subject by attaching themselves to the Subject.

When there is a state change in the Subject that the Observer may be interested in, a notify message is set, which calls the update method in each Observer.

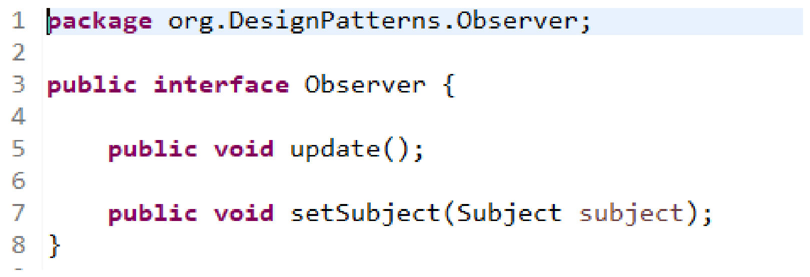
Let us look at an example of an Observer Design Pattern Implementation for Twitter Accounts



```java
package org.DesignPatterns.Observer;

public interface Subject {

    public void GenuineTwitterAccount(Observer observer);

    public void notifyAllAccounts();

    public void AnonymousTwitterAccount(Observer observer);

    public Object getNewTweets();

}
```

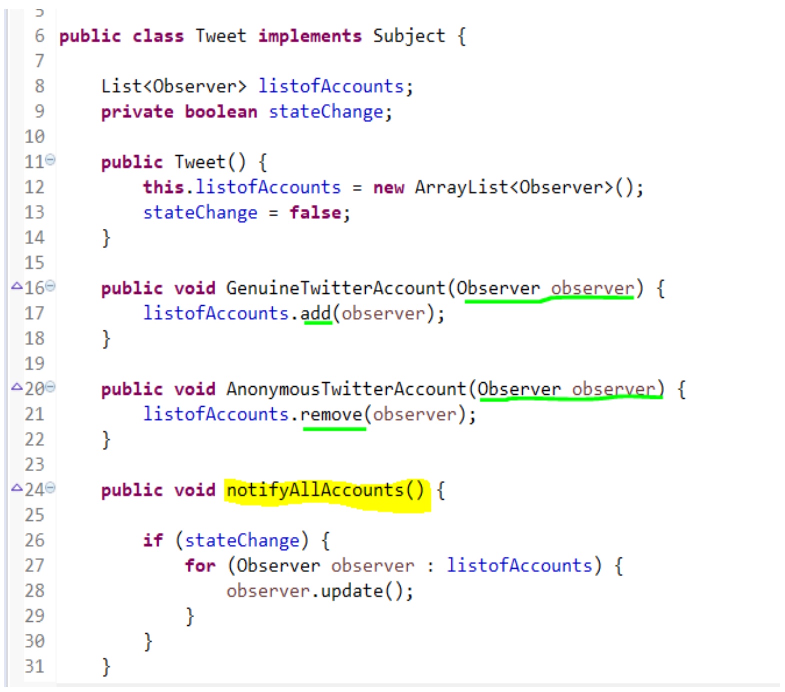
In the above code we see a Subject Interface that has two observers that can listen to the Object change, in this case they are the tweets being posted.

The observers are Genuine Twitter Accounts and Anonymous Twitter Accounts.



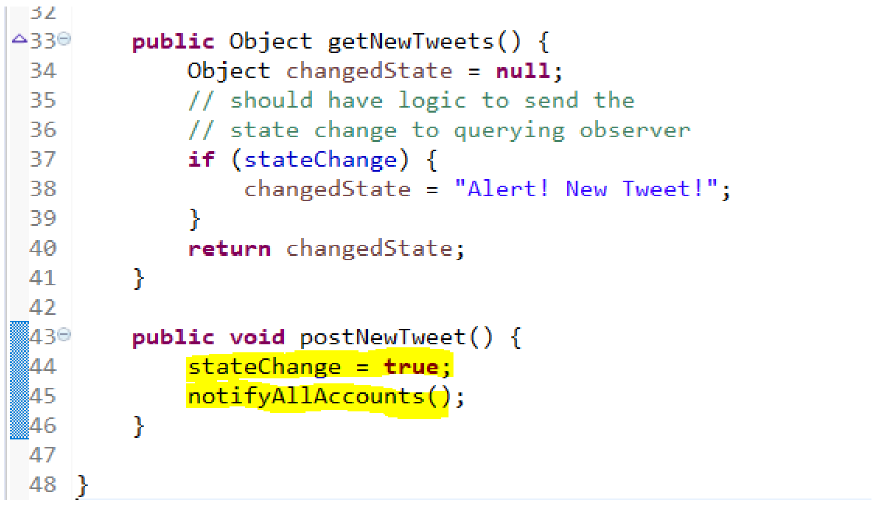
```java
package org.DesignPatterns.Observer;

public interface Observer {

    public void update();

    public void setSubject(Subject subject);
}
```

The above code is an Observer Interface that can update itself once it receives a notification that tweets have been posted (new state change of Object).



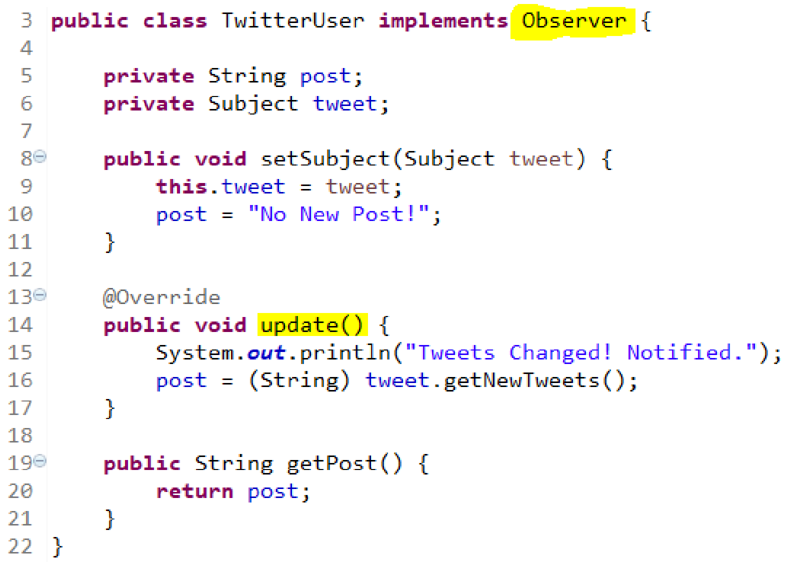
```java
public class Tweet implements Subject {

    List<Observer> listofAccounts;
    private boolean stateChange;

    public Tweet() {
        this.listofAccounts = new ArrayList<Observer>();
        stateChange = false;
    }

    public void GenuineTwitterAccount(Observer observer) {
        listofAccounts.add(observer);
    }

    public void AnonymousTwitterAccount(Observer observer) {
        listofAccounts.remove(observer);
    }

    public void notifyAllAccounts() {

        if (stateChange) {
            for (Observer observer : listofAccounts) {
                observer.update();
            }
        }
    }
```

The highlighted portion is the method that notifies all Accounts (all Observers) that there has been a new tweet (state change). The green marker portion indicates the Observers being added and removed. If the twitter accounts are genuine they are added and if they are anonymous they are removed but the subject does not know if the objects are genuine or anonymous. It only performs the action of adding and removing observers from its list.



```java
    public Object getNewTweets() {
        Object changedState = null;
        // should have logic to send the
        // state change to querying observer
        if (stateChange) {
            changedState = "Alert! New Tweet!";
        }
        return changedState;
    }

    public void postNewTweet() {
        stateChange = true;
        notifyAllAccounts();
    }

}
```

In the above code we can see that the when there is a new Tweet (state of the object) all accounts are notified.



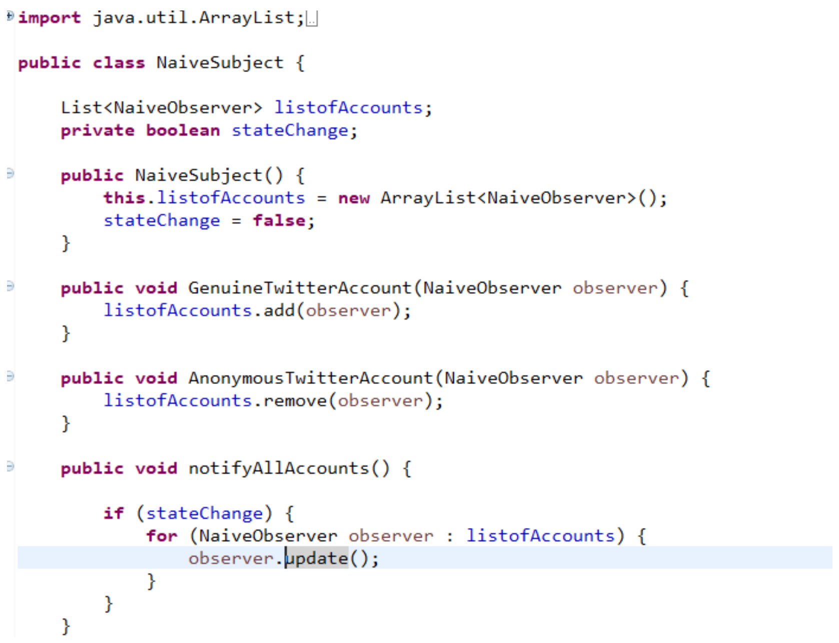
```java
public class TwitterUser implements Observer {

    private String post;
    private Subject tweet;

    public void setSubject(Subject tweet) {
        this.tweet = tweet;
        post = "No New Post!";
    }

    @Override
    public void update() {
        System.out.println("Tweets Changed! Notified.");
        post = (String) tweet.getNewTweets();
    }

    public String getPost() {
        return post;
    }
}
```

In the above code we can see that the User Account is implementing the Observer interface and the highlighted portion indicates how the accounts update itself based on the tweet being posted.

It indicates how the observer updates itself based on the new tweets it receives.

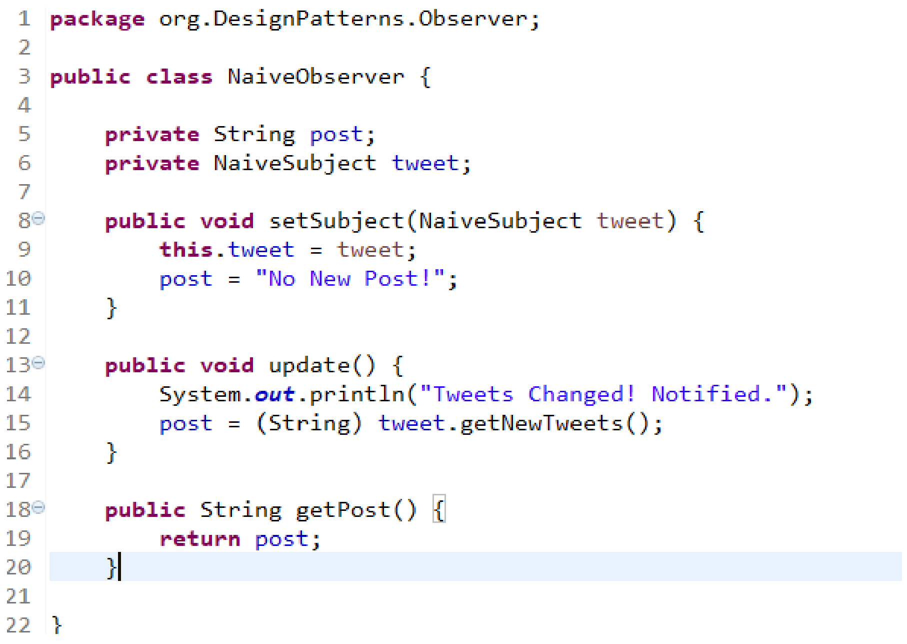
Now any number of User Accounts can implement this Observer interface when needed.

Let us look at an example where the Subject and observer are deeply coupled.



```java
import java.util.ArrayList;

public class NaiveSubject {

    List<NaiveObserver> listofAccounts;
    private boolean stateChange;

    public NaiveSubject() {
        this.listofAccounts = new ArrayList<NaiveObserver>();
        stateChange = false;
    }

    public void GenuineTwitterAccount(NaiveObserver observer) {
        listofAccounts.add(observer);
    }

    public void AnonymousTwitterAccount(NaiveObserver observer) {
        listofAccounts.remove(observer);
    }

    public void notifyAllAccounts() {

        if (stateChange) {
            for (NaiveObserver observer : listofAccounts) {
                observer.update();
            }
        }
    }
```

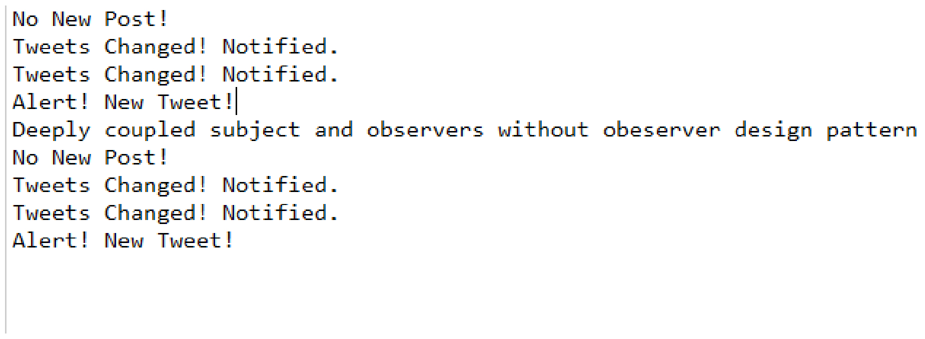
The above code is a Concrete Subject class that is deeply coupled to the Observer Class. It defeats the main purpose of this behavioral pattern.



```java
package org.DesignPatterns.Observer;

public class NaiveObserver {

    private String post;
    private NaiveSubject tweet;

    public void setSubject(NaiveSubject tweet) {
        this.tweet = tweet;
        post = "No New Post!";
    }

    public void update() {
        System.out.println("Tweets Changed! Notified.");
        post = (String) tweet.getNewTweets();
    }

    public String getPost() {
        return post;
    }

}
```

An Observer class that is deeply coupled to it’s subject class. If one of them breaks, then so does the other one.

There is not much of a difference in the output of the code or with how an Observer listens to a Subject, but it is a poor design when there is high coupling between one module and another.



```
No New Post!
Tweets Changed! Notified.
Tweets Changed! Notified.
Alert! New Tweet!
Deeply coupled subject and observers without obeserver design pattern
No New Post!
Tweets Changed! Notified.
Tweets Changed! Notified.
Alert! New Tweet!
```

The Observer design pattern is implemented in the ActionListner interface in JAVA. There is also a java.util.Observable that can be extended for any implementation.

Template Design Pattern

What is Template Design Pattern?

It defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure. Base class declares algorithm placeholder's and derived classes implement the placeholder.

The template pattern is known as a behavioral pattern as its used to manage algorithms, relationships and responsibilities between Objects. In Template pattern, an abstract class exposed defined way(s) or template(s) to execute its methods.

Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

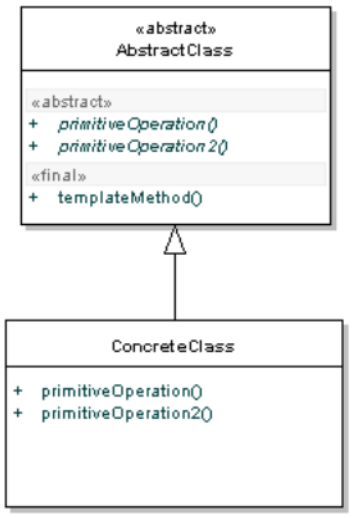
This pattern comes under the behavior pattern category.

Why Do we need Template Design Pattern?

A preset format that can be re-used instead of re-creating it each time.
A template can be defined which is then built on further variations. Template method can be used to build clean code. It's most evidently focused on avoiding duplications and let subclasses implement the behavior by overriding.

How the Template Design Pattern Works?

Structure of the Template Design Pattern.



[14]Fig 3

The structure of this design pattern is very simple. The templateMethod() should be made final and cannot be overridden.

All the operations used by this template method are made abstract, so their implementation is deferred to the sub classes.

Concrete class implements all the operations required by the template method, that were defined as abstract in the parent class.

There are four different types of methods used in the parent class (abstract class).

Concrete Methods, Abstract Methods, Hook Methods and Template Methods.

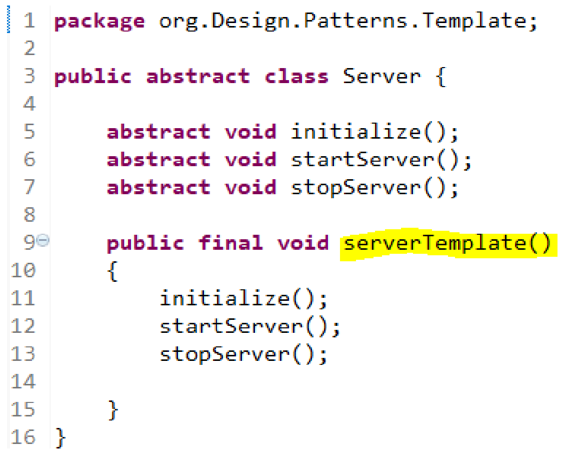
Concrete Methods: Methods that are fully complete and can be used by subclasses. They are utility methods.

Abstract Methods: Methods containing no implementation that must be implemented in sub classes.

Hook Methods: Methods containing a default implementation that may be overridden in some classes. They are intended to be overridden.

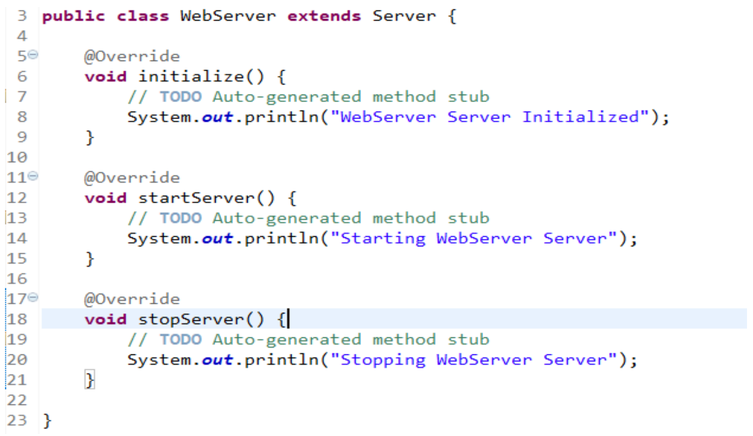
Template Methods: A method that calls any of the methods listed above in order to describe the algorithm without needing to implement the details.

Let us look at the following example where two different servers are able to invoke a template which has the methods that are used by both servers.



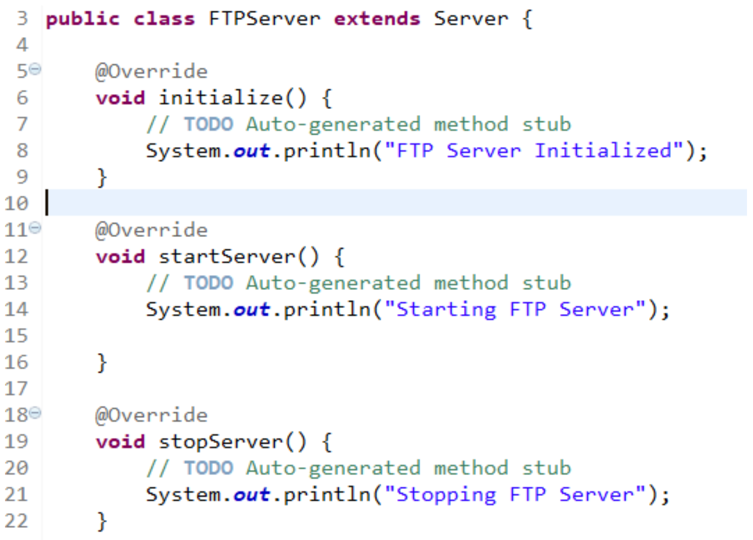
```java
package org.Design.Patterns.Template;

public abstract class Server {

    abstract void initialize();
    abstract void startServer();
    abstract void stopServer();

    public final void serverTemplate()
    {
        initialize();
        startServer();
        stopServer();

    }
}
```

The highlighted portion of the code is the template method in the parent class Server that will be used by the subclasses that inherit this class.



```java
public class WebServer extends Server {

    @Override
    void initialize() {
        // TODO Auto-generated method stub
        System.out.println("WebServer Server Initialized");
    }

    @Override
    void startServer() {
        // TODO Auto-generated method stub
        System.out.println("Starting WebServer Server");
    }

    @Override
    void stopServer() {
        // TODO Auto-generated method stub
        System.out.println("Stopping WebServer Server");
    }

}
```

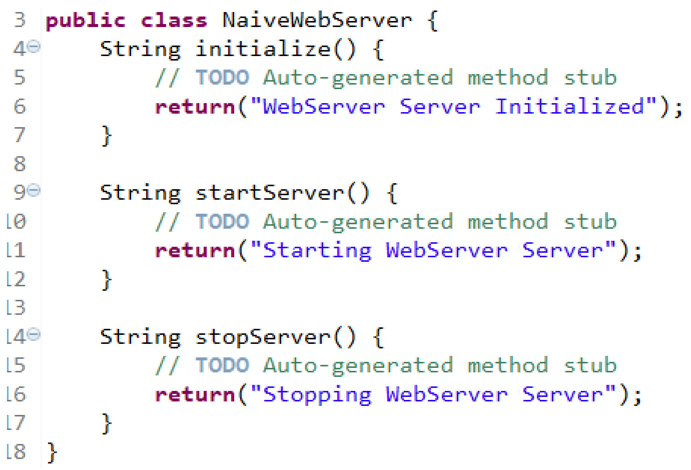
In the above code A Web Server inherits the abstract operations that are also present in the template() method.



```java
public class FTPServer extends Server {

    @Override
    void initialize() {
        // TODO Auto-generated method stub
        System.out.println("FTP Server Initialized");
    }

    @Override
    void startServer() {
        // TODO Auto-generated method stub
        System.out.println("Starting FTP Server");

    }

    @Override
    void stopServer() {
        // TODO Auto-generated method stub
        System.out.println("Stopping FTP Server");
    }
```

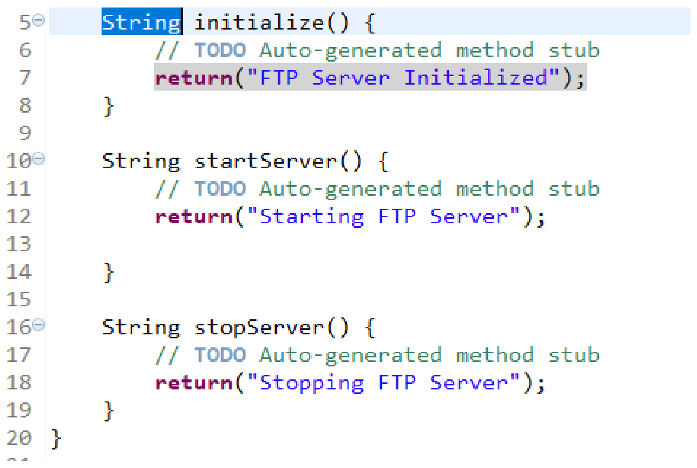
In the above code a FTP Server class inherits the abstract methods which are also in the template method() and overrides it with its own definition.

Let us look at a naïve implementation of the servers without using template method design pattern.



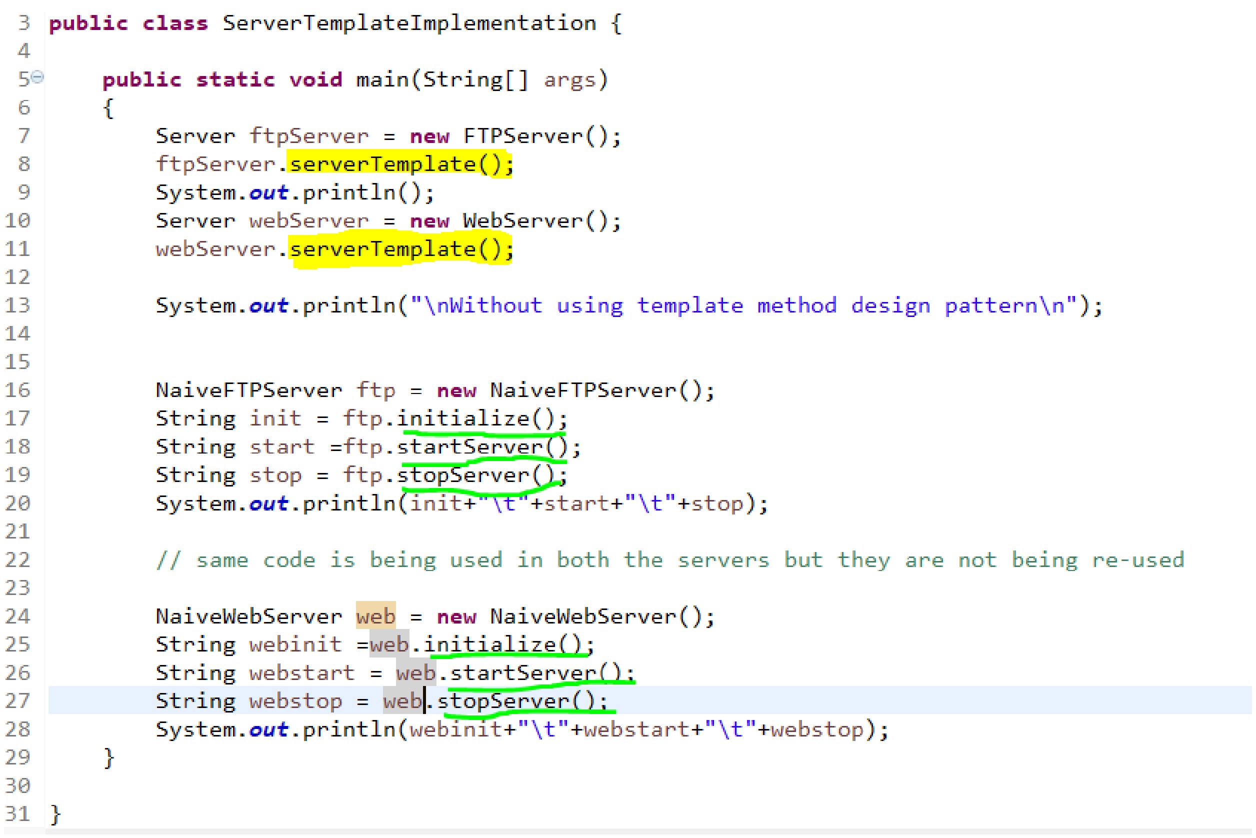
```java
public class NaiveWebServer {
    String initialize() {
        // TODO Auto-generated method stub
        return("WebServer Server Initialized");
    }

    String startServer() {
        // TODO Auto-generated method stub
        return("Starting WebServer Server");
    }

    String stopServer() {
        // TODO Auto-generated method stub
        return("Stopping WebServer Server");
    }
}
```

A Webserver that implements methods without using the template method design pattern in it's server implementation module.



```java
    String initialize() {
        // TODO Auto-generated method stub
        return("FTP Server Initialized");
    }

    String startServer() {
        // TODO Auto-generated method stub
        return("Starting FTP Server");

    }

    String stopServer() {
        // TODO Auto-generated method stub
        return("Stopping FTP Server");
    }
}
```

A FTP server that implements the same methods as the Web Server. It is evident that there is code duplication and code re-usability is not achieved.

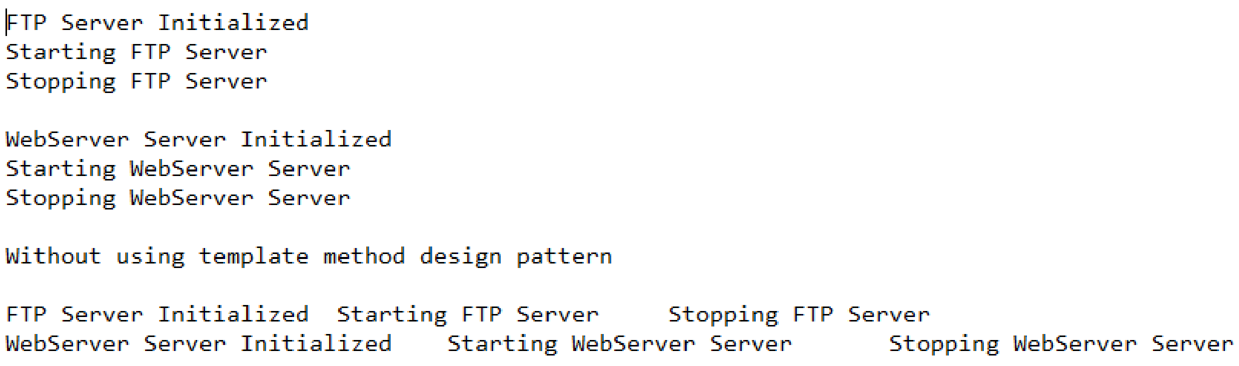


```java
public class ServerTemplateImplementation {

    public static void main(String[] args)
    {
        Server ftpServer = new FTPServer();
        ftpServer.serverTemplate();
        System.out.println();
        Server webServer = new WebServer();
        webServer.serverTemplate();

        System.out.println("\nWithout using template method design pattern\n");


        NaiveFTPServer ftp = new NaiveFTPServer();
        String init = ftp.initialize();
        String start =ftp.startServer();
        String stop = ftp.stopServer();
        System.out.println(init+"\t"+start+"\t"+stop);

        // same code is being used in both the servers but they are not being re-used

        NaiveWebServer web = new NaiveWebServer();
        String webinit =web.initialize();
        String webstart = web.startServer();
        String webstop = web.stopServer();
        System.out.println(webinit+"\t"+webstart+"\t"+webstop);
    }

}
```

In the above piece of code, we can see that the ftp Server makes one call to the template method and so does the Webserver. This showcases simplicity in implementing methods that can be re-used.

In the portions marked with green markers, the same methods are called by both the servers which showcases code duplication which is not a very efficient design.

The output of the code is the same but it is clear that using template method design pattern is more efficient in places where code can be re-used.



```
FTP Server Initialized
Starting FTP Server
Stopping FTP Server

WebServer Server Initialized
Starting WebServer Server
Stopping WebServer Server

Without using template method design pattern

FTP Server Initialized	Starting FTP Server	Stopping FTP Server
WebServer Server Initialized	Starting WebServer Server	Stopping WebServer Server
```

It is very commonly used and in JAVA it can be seen in the Arrays class where it is used for sorting and in the JFRAME where the update() method is used as a template method.

REFERENCES

[1] https://dzone.com/articles/design-patterns-iterator
[2]https://javapapers.com/design-patterns/iterator-design-pattern/
[3] https://www.journaldev.com/1716/iterator-design-pattern-java
[4]https://sourcemaking.com/design_patterns/iterator
[5]https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html
[6]http://www.oodesign.com/iterator-pattern.html
[7]http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer
[8]https://sourcemaking.com/design_patterns/iterator/java/1
[9]https://dzone.com/articles/design-patterns-uncovered
[10]https://www.journaldev.com/1827/java-design-patterns-example-tutorial
[11] https://dzone.com/articles/design-patterns-uncovered
[12] https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
[13] http://www.oodesign.com/observer-pattern.html
[14] https://dzone.com/articles/design-patterns-template-method
[15] https://sourcemaking.com/design_patterns/template_method
[16] https://javapapers.com/design-patterns/template-method-design-pattern
[17] http://www.oodesign.com/template-method-pattern.html
[18] https://www.tutorialspoint.com/design_pattern/template_pattern.htm