Observer Design Pattern in Java

journaldev.com/1739/observer-design-pattern-in-java

July 28, 2013

Observer Pattern is one of the **behavioral design pattern**. Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

Observer Design Pattern

According to GoF, observer design pattern intent is;

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Subject contains a list of observers to notify of any change in it's state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update.

Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

Java provides inbuilt platform for implementing Observer pattern through *java.util.Observable* class and *java.util.Observer* interface. However it's not widely used because the implementation is really simple and most of the times we don't want

to end up extending a class just for implementing Observer pattern as java doesn't provide multiple inheritance in classes.

Java Message Service (JMS) uses **Observer design pattern** along with **Mediator pattern** to allow applications to subscribe and publish data to other applications.

Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

Observer Pattern Java Example

For our observer pattern java program example, we would implement a simple topic and observers can register to this topic. Whenever any new message will be posted to the topic, all the registers observers will be notified and they can consume the message.

Based on the requirements of Subject, here is the base Subject interface that defines the contract methods to be implemented by any concrete subject.

```
package com.journaldev.design.observer;

public interface Subject {

 //methods to register and unregister observers
 public void register(Observer obj);
 public void unregister(Observer obj);

 //method to notify observers of change
 public void notifyObservers();

 //method to get updates from subject
 public Object getUpdate(Observer obj);

}
```

Next we will create contract for Observer, there will be a method to attach the Subject to the observer and another method to be used by Subject to notify of any change.

```
package com.journaldev.design.observer;

public interface Observer {

 //method to update the observer, used by subject
 public void update();

 //attach with subject to observe
 public void setSubject(Subject sub);
}
```

Now our contract is ready, let's proceed with the concrete implementation of our topic.

```
package com.journaldev.design.observer;

import java.util.ArrayList;
import java.util.List;

public class MyTopic implements Subject {

 private List<Observer> observers;
 private String message;
 private boolean changed;
 private final Object MUTEX= new Object();

 public MyTopic(){
  this.observers=new ArrayList<>();
 }
 @Override
 public void register(Observer obj) {
  if(obj == null) throw new NullPointerException("Null Observer");
  synchronized (MUTEX) {
  if(!observers.contains(obj)) observers.add(obj);
  }
 }

 @Override
 public void unregister(Observer obj) {
  synchronized (MUTEX) {
  observers.remove(obj);
  }
 }

 @Override
 public void notifyObservers() {
  List<Observer> observersLocal = null;
  //synchronization is used to make sure any observer registered after message is received is
not notified
  synchronized (MUTEX) {
   if (!changed)
    return;
   observersLocal = new ArrayList<>(this.observers);
   this.changed=false;
  }
  for (Observer obj : observersLocal) {
   obj.update();
  }

 }

 @Override
 public Object getUpdate(Observer obj) {
  return this.message;
 }

 //method to post message to the topic
```

```
 public void postMessage(String msg){
  System.out.println("Message Posted to Topic:"+msg);
  this.message=msg;
  this.changed=true;
  notifyObservers();
 }

}
```

The method implementation to register and unregister an observer is very simple, the extra method is *postMessage()* that will be used by client application to post String message to the topic. Notice the boolean variable to keep track of the change in the state of topic and used in notifying observers. This variable is required so that if there is no update and somebody calls *notifyObservers()* method, it doesn't send false notifications to the observers.

Also notice the use of synchronization in *notifyObservers()* method to make sure the notification is sent only to the observers registered before the message is published to the topic.

Here is the implementation of Observers that will watch over the subject.

```
package com.journaldev.design.observer;

public class MyTopicSubscriber implements Observer {

 private String name;
 private Subject topic;

 public MyTopicSubscriber(String nm){
  this.name=nm;
 }
 @Override
 public void update() {
  String msg = (String) topic.getUpdate(this);
  if(msg == null){
   System.out.println(name+":: No new message");
  }else
  System.out.println(name+":: Consuming message::"+msg);
 }

 @Override
 public void setSubject(Subject sub) {
  this.topic=sub;
 }

}
```

Notice the implementation of *update()* method where it's calling Subject *getUpdate()* method to get the message to consume. We could have avoided this call by passing message as argument to *update()* method.

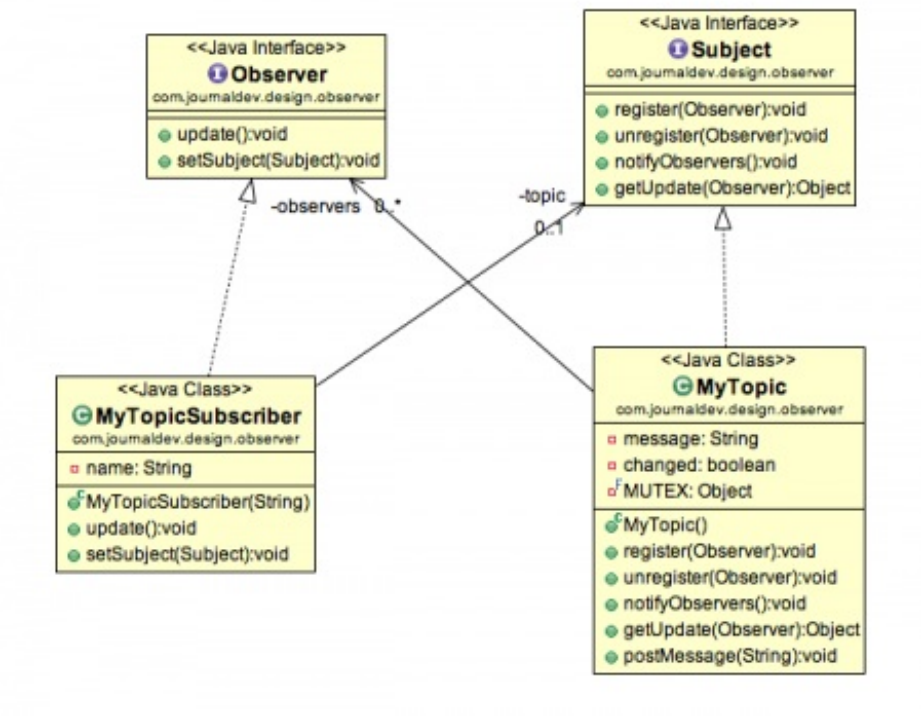
Here is a simple test program to consume our topic implementation.

```
package com.journaldev.design.observer;

public class ObserverPatternTest {

 public static void main(String[] args) {
  //create subject
  MyTopic topic = new MyTopic();

  //create observers
  Observer obj1 = new MyTopicSubscriber("Obj1");
  Observer obj2 = new MyTopicSubscriber("Obj2");
  Observer obj3 = new MyTopicSubscriber("Obj3");

  //register observers to the subject
  topic.register(obj1);
  topic.register(obj2);
  topic.register(obj3);

  //attach observer to subject
  obj1.setSubject(topic);
  obj2.setSubject(topic);
  obj3.setSubject(topic);

  //check if any update is available
  obj1.update();

  //now send message to subject
  topic.postMessage("New Message");
 }

}
```

When we run above program, we get following output.

```
Obj1:: No new message
Message Posted to Topic:New Message
Obj1:: Consuming message::New Message
Obj2:: Consuming message::New Message
Obj3:: Consuming message::New Message
```

Java Observer Pattern Class Diagram



Observer design pattern is also called as publish-subscribe pattern. Some of it's implementations are;

- java.util.EventListener in Swing
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener

That's all for Observer design pattern in java, I hope you liked it. Share your love with comments and by sharing it with others.