Observer Design Pattern in Java

APRIL 2, 2018 BY PANKAJ — 37 COMMENTS

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);
                }
        }
```

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.

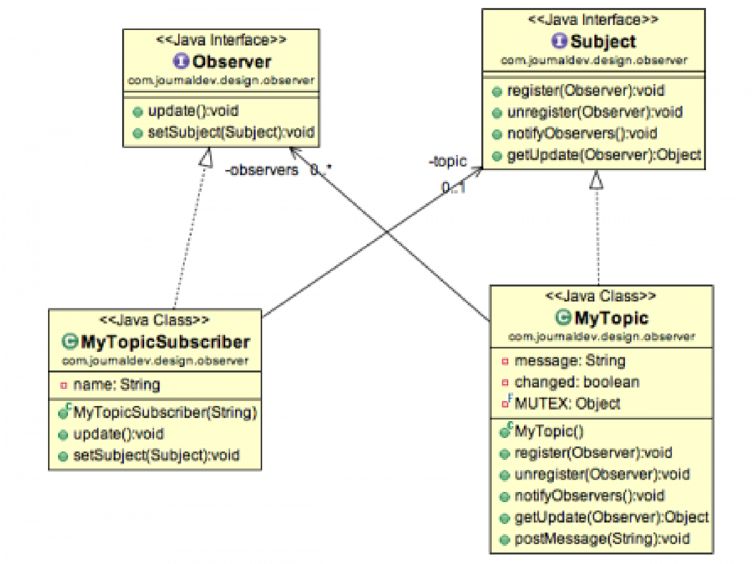
Here is a simple test program to consume our topic implementation.

```
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);
```

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.

« PREVIOUS
Memento Design Pattern in Java

NEXT »
State Design Pattern in Java

About Pankaj
If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

John says
MARCH 9, 2018 AT 9:41 AM
You said that "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."
How do we decide which implementation to use ? I am guessing that if some of the observers only care that update has happened, but don't care what was the update, then just update() is useful. If some are interested in what was updated, then update(Object updateInfo) makes sense.
Also, why are we passing observer inside getUpdate(Observer obj) ? Maybe observable can use that to track which all observers got the message ?
Reply

John says
MARCH 9, 2018 AT 9:19 AM
It would be helpful to put a comment for why public void unregister(Observer obj) has a synchronized block, It is because you cannot add elements to arraylist and also remove elements from it at the same time. It will give a java.util.ConcurrentModificationException.
Reply

Anand says
JANUARY 23, 2018 AT 4:46 PM
Really good article. Much better than others which are out there
Reply

Pankaj says
JANUARY 23, 2018 AT 8:33 PM
Thanks for the appreciation Anand.
Reply

Krishna says
JANUARY 2, 2018 AT 12:10 PM
Awesome article, such a simple and easy one to understand it.
Reply

Faizuddin Ahmed Shaik says
JANUARY 3, 2017 AT 8:42 PM
I think this code has a flaw. If we set the message once and then do not set again and then check if there is any update, then according to the code the old msg is not null and it will tell that the update actually happened.
Reply

GT says
MAY 7, 2017 AT 7:09 AM
Agreed.
Reply

Shan says
JUNE 18, 2018 AT 7:32 AM
Yeah as per code your comment is valid. But please look into that the update method will be called only if the notify method called. This will happen only if new message is set.. So as per the pattern design this not seems like a flaw.
Reply

Subhash Kumar says
OCTOBER 27, 2016 AT 1:36 AM
Crystal clear. Useful for both the beginners and advanced level developers.
Reply

Ankit says
JULY 16, 2016 AT 8:08 AM
Thanks for the nice article. But I wanted to check/know the reason why synchronization is required in this example? Is it not an overkill?
Reply

Sumit says
JUNE 19, 2016 AT 11:45 AM
Thanks for such a good explanation. Keep up the good work.
Reply

Jason Alls says
JUNE 16, 2016 AT 1:43 AM
Great article, exactly what I was looking for. Thank you, and keep up the good work.
Reply

Pankaj says
JUNE 16, 2016 AT 2:47 AM
Glad you liked it Jason.
Reply

Ivan says
MAY 8, 2016 AT 6:10 PM
Hi, the method does not use the passed in variable: Observer obj
public Object getUpdateMsg(Observer obj)

Is it necessary to rewrite it as 'public Object getUpdateMsg()' ?
Reply

Khosro Makari says
MARCH 31, 2016 AT 1:40 PM
Thanks Mr Pankaj. I read design pattern from multiple sites, but your tutorial was more useful and was best. keep doing your best!
Reply

sridhar says
FEBRUARY 22, 2016 AT 9:20 PM
Very good explanation
Reply

Biswajit Mohapatra says
JANUARY 4, 2016 AT 4:11 AM
Excellent explanation and it helped me lot to understand what is Observer design pattern. Thanks a lot.
Reply

Darpan says
MAY 11, 2015 AT 5:36 AM
After spending so much time on many tutorials, this was the best. Great explaination.
Reply

Oleg says
MAY 9, 2015 AT 12:14 AM
Hi, Pankaj!! Thx for tutorials, like them much. Bur this one seems to be confusing a little. Here is more clear one. Implement this, if you like it.
Reply

Karan says

APRIL 11, 2015 AT 9:31 AM

Hi,
I would like to know whether same implementation can be used for stocks exchange ,
that is there are stream of stocks are coming in and the clients are waiting on the other side to see the current stock value of a particular company

Reply

Jasvinder Singh says

FEBRUARY 26, 2015 AT 8:38 AM

The register() in MyTopic class should also call the Observer's setSubject() passing 'this' as a parameter thereby abstracting, the attaching of the subject to the observer, from the client code. The client code should only need to register the observer to the topic.
Importance of passing observer as a parameter to the MyTopic's getUpdate() can be demonstrated by having a call back method in Observer interface.

Reply

yuval says

FEBRUARY 23, 2015 AT 4:33 AM

i didn't understand why the method getUpdate(Obserever obj) needs the Observer reference parameter since it doesnt uses it.

Reply

md farooq says

JANUARY 2, 2015 AT 10:21 AM

Very good and simple explanation.

Reply

Prince says

SEPTEMBER 30, 2014 AT 10:50 PM

How to run this program using netbeans. Please explain it.

Reply

Gerardo says

JULY 22, 2014 AT 3:38 PM

Thanks for this explanation!
The only thing I don't completely get is the need of the `setSubject` method in the Observer. Could you please explain?
Reply

Akash kumar says
AUGUST 5, 2014 AT 1:40 AM
Hi Gerardo
The object of an observer needs a subject to monitor, so the observer object should have a method to assign that subject to it. So that whenever a change occurs in the subject the observer gets to know it. So is the need of that setSubject method.
Reply

Nikhil Verma says
SEPTEMBER 3, 2016 AT 8:32 AM
Instead of having a setSubject method on the Observer side, would it not be better if the subject is simply passed as an argument in the update method of the observer .
Something like update(Subject sub), wherein this would be passed as argument from the subject's side while calling the method.
Reply

Tanaya Karmakar says
DECEMBER 4, 2016 AT 9:06 AM
Hi , there are two ways in which observers are getting updated , one is push and the other is pull , the method u mentioned is push method but pull method is considered better since observers are not forced to take the things they don't need , in pull method observers pull the things they need from subject
Reply

Ashish Patel says
JUNE 10, 2014 AT 4:43 AM
Very Good Explanation !
Reply

mohammed says

MARCH 21, 2014 AT 11:28 PM

Thanks for your effort i really learned from it but your line (this.observers=new ArrayList();)

it need to enhanced to (this.observers = new ArrayList();)

Thanks,

Mohammed Gamal

Software Engineer

Reply

Savi says

FEBRUARY 16, 2014 AT 11:07 AM

Very well explained. Thanks…

Reply

Vinh Bui says

FEBRUARY 11, 2014 AT 7:17 AM

Thank for your explanation. But I still have a confusing.

As you said: "//synchronization is used to make sure any observer registered after message is received is not notified" in comment of class my topic, the synchronization is used for that purpose.

But as my understanding about synchronization, the method "public void register(Observer obj)" and method "public void notifyObservers() " still can be run parallel.

It means the observers that registered after message is received also can be notified.

Reply

Pankaj says

FEBRUARY 12, 2014 AT 12:12 AM

Very nice observation, yes you are correct.

I have updated the register and deregister methods to use synchronization on MUTEX object, now when message is getting sent to the registered observers, it will not be sent to any observer registered after the message is received.

Reply

sushil says

JANUARY 17, 2014 AT 7:23 AM

Explanation is very good

Reply

Jeni says

JANUARY 3, 2014 AT 4:53 PM

Thanks for your post. You describe all topics very well.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

GET IT ON Google Play

DESIGN PATTERNS TUTORIAL

RECOMMENDED TUTORIALS

- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial