Observer Design Pattern

In observer design pattern multiple observer objects registers with a subject for change notification. When the state of subject changes, it notifies the observers. Objects that listen or watch for change are called observers and the object that is being watched for is called subject.

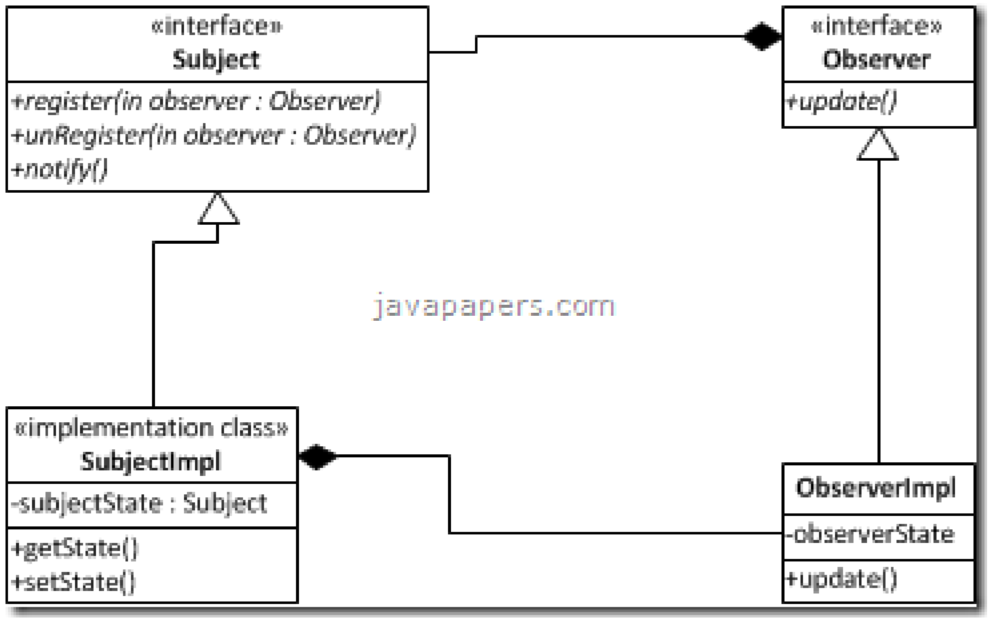
Pattern involved is also called as publish-subscribe pattern. Model view controller (MVC) architecture’s core uses the observer design pattern.

Important Points on Observer Design Pattern

- Subject provides interface for observers to register and unregister themselves with the subject.
- Subject knows who its subscribers are.
- Multiple observers can subscribe for notifications.
- Subject publishes the notifications.
- Subject just sends the notification saying the state has changed. It does not pass any state information.
- Once the notification is received from subject, observers call the subject and get data that is changed.

The above last two points are not strictly followed in observer design pattern implementation. Along with the notification, state is also passed in some implementation so that the observer need not query back to know the status. It is better not to do this way.

Observer Design Pattern UML

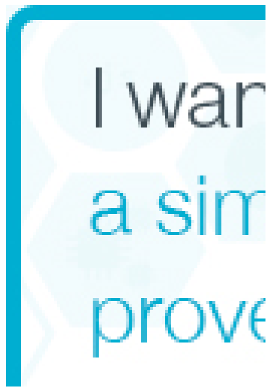


Scenario for Observer Pattern Usage

- When multiple objects depend on state of one object and all these multiple objects should be in sync with the state of this one object then observer design pattern is the right choice to apply.
- Consider an excel sheet, data is shown to the user in different views. Generally data is is shown in grid cells and as required different graphs, charts can be created for same data. Underlying data is same and when that data (subject) state changes all the different view are updated.

Observer and Observable Java API in jdk

JDK provides Observer and Observable classes as part of util package. This is not something new and these classes were available since JDK 1.0. Observer is an interface which needs to be implemented to observe the state change in a observable subject. Observable is a class which should be extended by a subject. Observable provides implementation for methods to register or unregister an Observer and to notify the Observer objects.



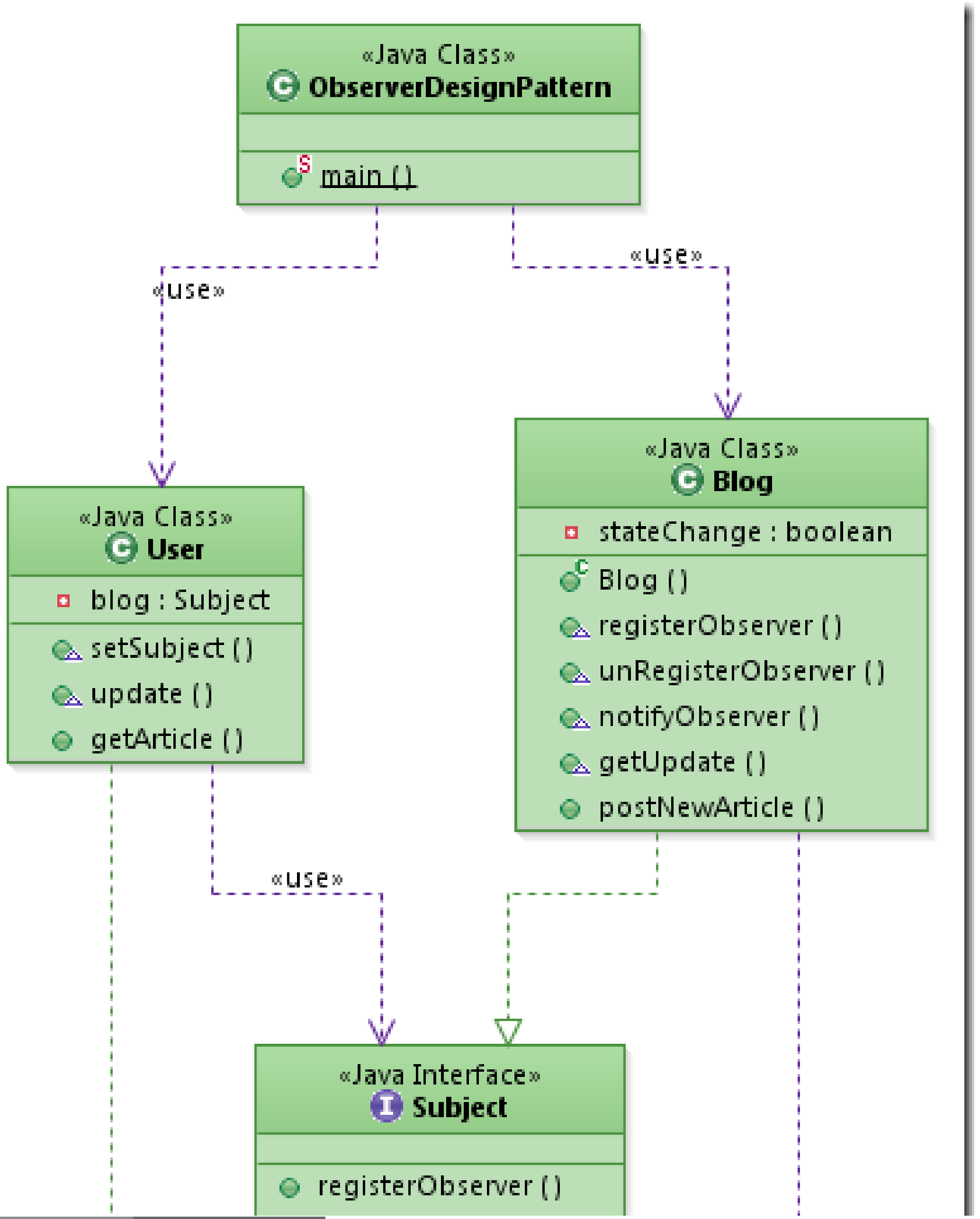
Using these APIs, Observable and Observer in real time application may not be possible most of time. As we all know, java does not support multiple inheritance. I can extend only one class and I do not want to exhaust that option with this Observable class. In a real time application I may need that option to be used for business inheritance.

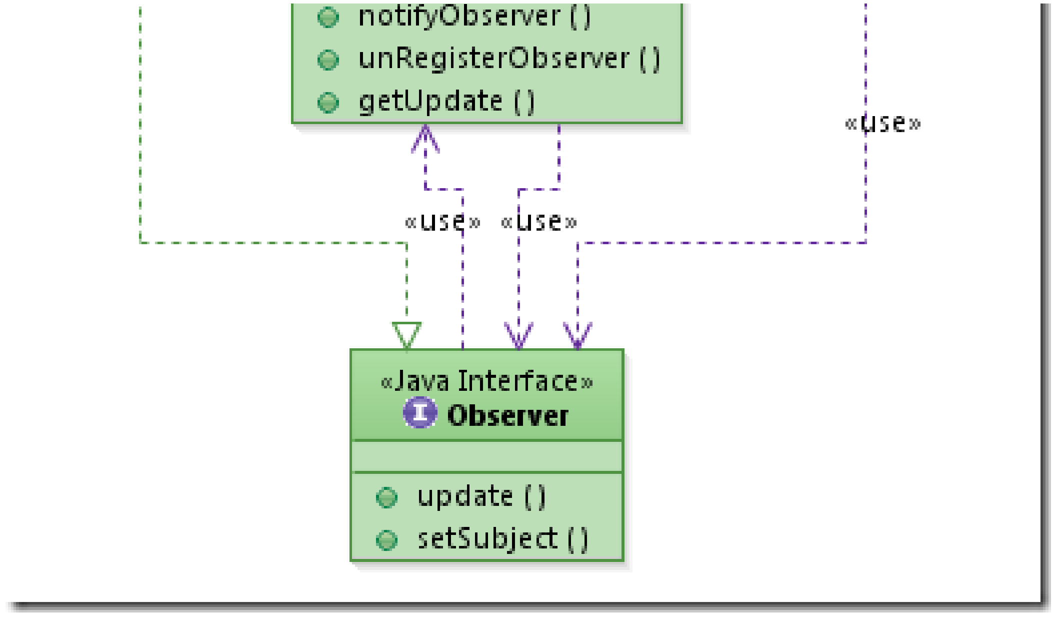
Prefer composition over inheritance is a golden rule. So I thought of using Observable in composition than inheritance. That option is also ruled out because *setChanged()* method has access specified as *protected*. So as per Java access speficier rules, protected attributes cannot be accessed from outside the package. So the only option left is to do a custom implementation for Observer pattern. It is not a tedious one.

Example Observer Design Pattern

Let us take a blog and subscriber example for observer design pattern sample implementation. Assume that there is a blog and users register to that blog for update. When a new article is posted in the blog, it will send update to the registered users saying a new article is posted. Then the user will access the blog and read the new article posted. In this example, blog is the subject and user is the observer.

UML for Example





Observer.java

```
package com.javapapers.designpattern.observer;

public interface Observer {

        public void update();

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

User.java

```
package com.javapapers.designpattern.observer;
```

```java
public class User implements Observer {

        private String article;
        private Subject blog;

        public void setSubject(Subject blog) {
                this.blog = blog;
                article = "No New Article!";
        }

        @Override
        public void update() {
                System.out.println("State change reported by Subject.");
                article = (String) blog.getUpdate();
        }

        public String getArticle() {
                return article;
        }
}
```

Subject.java

```java
package com.javapapers.designpattern.observer;

public interface Subject {
        public void registerObserver(Observer observer);

        public void notifyObserver();

        public void unRegisterObserver(Observer observer);

        public Object getUpdate();

}
```

Blog.java

```java
package com.javapapers.designpattern.observer;

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

public class Blog implements Subject {

        List observersList;
        private boolean stateChange;

        public Blog() {
                this.observersList = new ArrayList();
                stateChange = false;
        }
        public void registerObserver(Observer observer) {
                observersList.add(observer);
        }

        public void unRegisterObserver(Observer observer) {
                observersList.remove(observer);
        }

        public void notifyObserver() {

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

ObserverDesignPattern.java

```
package com.javapapers.designpattern.observer;

public class ObserverDesignPattern {
        public static void main(String args[]) {
                Blog blog = new Blog();
                User user1 = new User();
                User user2 = new User();
                blog.registerObserver(user1);
                blog.registerObserver(user2);
                user1.setSubject(blog);
                user2.setSubject(blog);

                System.out.println(user1.getArticle());
                blog.postNewArticle();
                System.out.println(user1.getArticle());
        }

}
```

Output

```
No New Article!
State change reported by Subject.
State change reported by Subject.
Observer Design Pattern
```

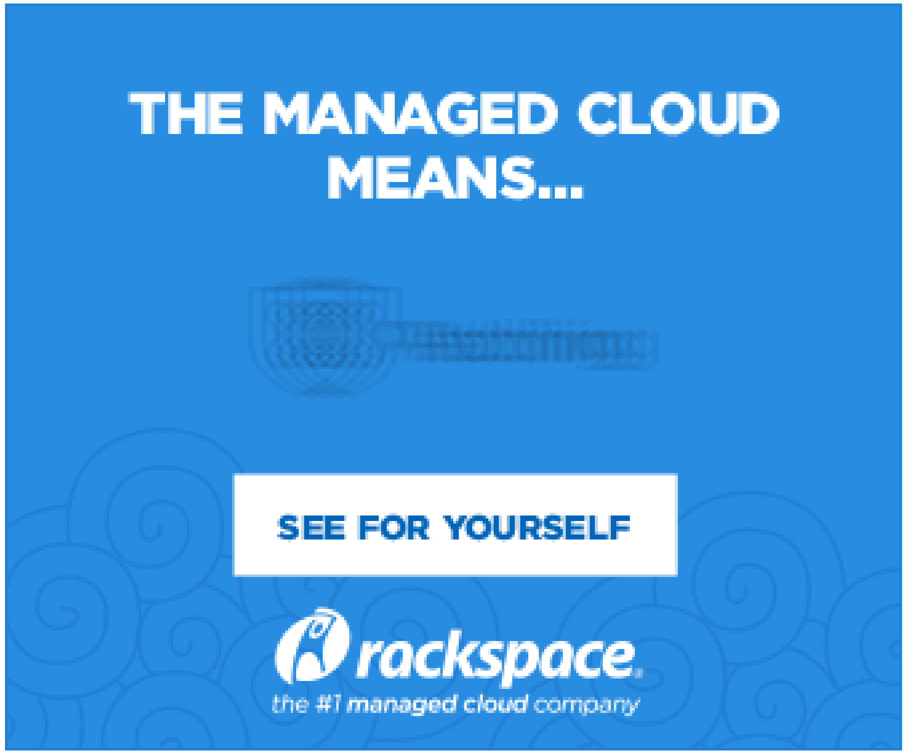
Usage of Observer Design Pattern in Java API

HttpSessionBindingListener is an example where Observer design pattern in used in Java API.

Download Source Code for Observer Design Pattern

Observer Design Pattern Source Code

This Behavioral Design Pattern tutorial was added on 26/03/2013.



« Android SQLite Database

Ant Colony Optimization in Java »

Comments on “Observer Design Pattern”

Tom *26/03/2013 at 1:49 pm*

Could you post the name of the program used to create the UML diagram?

Reply

Chiranjeevi *01/04/2013 at 3:22 pm*

even i need to know which program you used to generate UML diagram..

Reply

Gaurav *07/04/2013 at 2:16 pm*

Hey Tom and Chiranjeevi,

I would also like to know which software was used to generate the class diagrams. But, here is what I use. And If you too use eclipse, you'll love it.
http://www.objectaid.com/

Reply

Joe *27/07/2013 at 3:54 pm*

Looks good at first look. Thanks Gaurav, let me try this.

Reply

Joe *27/07/2013 at 3:47 pm*

I use StarUML, MS Visio for UML diagrams.

Reply

Anonymonus *26/03/2013 at 3:38 pm*

Can You please Explain the Steps in which we should proceed for Getting better grip in Java

Reply

Joe *29/03/2013 at 3:34 pm*

Reading through one good java book is the first step. It is important to learn lots and lots of theory.

Second step is to practice code. Apply everything you learn in that project by adding features to it.

Reply

Vishal *26/03/2013 at 7:16 pm*

Nice explanation.

Thanks Joe.

Reply

shendiao *27/03/2013 at 11:24 am*

Nice as always.

Reply

shendiao *28/03/2013 at 7:11 am*

Hi Joe,

You mentioned:”

The above last two points are not strictly followed in observer design pattern implementation. Along with the notification, state is also passed in some implementation so that the observer need not query back to know the status. It is better not to do this way.”

Why is it not better to do that way? In this case, I think the “Observers” don’t need to know the specific subject, isn’t it a better decoupling way?

Reply

Joe *29/03/2013 at 3:31 pm*

Shendiao,

That is a nice question and it shows that you have keenly observed the pattern.

We cannot achieve decoupling because of only this. Since already we require subject’s reference in observer for registration. Registration is initiated from observer side. In the example I have shown, I have not used register(this), so it may dilute this point. But, I have used the client (main method) to make the registration call, so indirectly there is coupling.

Second, why I prefer observer to make the call is, it gives more flexibility. Subject can expose different interface abstracting details and observer can choose the interface as of its choice. Subject passing the complete reference of itself is I feel like standing naked on top of clock tower. (hey no offence meant, just making it lighter..)

Hope I have made it clear, Shendio thats a nice question and thanks for asking it.

Reply

Gaurav *07/04/2013 at 2:12 pm*

Hi Shendiao,
Very interesting question indeed. I myself was in doubt when I first got to know about the observer pattern.

But, in case you have stuck to the basic design practices, you'll be fine.

1. Data Hiding – Although the reference of 'Subject' has to be passed, but if the 'Subject' has exposed just the right information, things will not foul up.

2. Coupling – Yes, you do have to compromise a bit here. But, when we talk about more than one option for a given service (blog), we turn to the following solutions:

a) static references of all the services. e.g. using Enum for all service implementations.

b) dynamic dependency injections, but that too uses static references internally.

3. regsiter(this) – Yes, I agree it gives a feeling of standing naked on the top of a building. But, its all about perspective.
Consider a scenario where a user wants to subscribe to a newspaper. There 'register(this)' could either be register(user) or simply register(user.getAddress()). Registering only the address makes more sense, as the newspaper has to be delivered to an address. And also it doesn't give a feeling of being naked anymore. :-)

Reply

---

jeevitha *15/04/2013 at 6:05 pm*

Hi joe,

Can u explain me how to sort the even position in ascending order and odd position in descending order in java

For Example:

Array = { 23,75,98,7,100}

output = {100,7,98,75,23}

Final output = {5,100,7,98,75,23}

5 represents the size of the array.

Reply

Anonymous *28/04/2013 at 6:38 pm*

simplest way is to segregate even position array and odd position array and sort them individually and then merge…else you can sort them without segregating by incrementing index twice rather than 1

Reply

---

sumit *07/06/2013 at 7:28 pm*

i did not understand it,please make it more clear.

Reply

Joe *27/07/2013 at 3:34 pm*

Which part of it Sumit, you are not able to understand? or whole of the concept?

Reply

Sharath *18/07/2013 at 4:34 pm*

Nice Example, this concept is clear.

Thanks joe.

Reply

Joe *27/07/2013 at 3:33 pm*

Welcome Sharath.

Reply

Raghav Agrawal *21/08/2013 at 12:37 am*

Hi Joe,

I did not understand the use of ‘stateChange’ variable used in the Observable implementation. Once flipped to true it is never reset to false.

That was a very minor comment but as a whole, its amazing how precise (and concise) your articles are. I love reading them.

Thanks,
Raghav

Reply

Dhiral Pandya *21/08/2013 at 1:28 pm*

Nice article. Here you are using two methods in Observer

public void update();
public void setSubject(Subject subject);

But can we use only one method

update(Subject subject);

It means in our Blog class we call it like this

```
for (Observer observer : observersList) {
observer.update(this);
}
```

Reply

Anonymous *18/09/2013 at 7:42 am*

Nice – very useful as always – thanks!

Reply

---

fred *25/09/2013 at 8:47 am*

Hi,

Thanks for the article. I was just wondering why you have defined interfaces? Particularly, at some point you implement interface, and then override the function? My question is that, if there is something in the interface that you will define later, why you define it in the firs place?

cheers,

fred

Reply

Andy *30/09/2013 at 5:28 pm*

thanks, nice tutorial

Reply

Joe *01/10/2013 at 11:38 am*

Welcome Andy.

Reply

Anonymous *28/10/2013 at 6:03 pm*

Can you please explain the difference between events concept and observer pattern.

Reply

kumar *20/02/2014 at 2:26 pm*

Hi joe,

Instead of notifying “state has changed” it is enough to pass state changed infomation if i use this what problem will i get

Reply

SC *09/06/2014 at 8:54 pm*

Why did you delete my post before Joe?

Reply

Joe *10/06/2014 at 11:00 am*

Your email looked like some spam emailid and instantly I have deleted it. Sorry, please post your comment again.

Reply

SC *12/06/2014 at 12:46 am*

Joe – My comment was that although this is a good tutorial. The names that you used for your variables do not make it very clear how the observer pattern works just from reading the code on this site. I had to copy and paste the code and run it to realize how the User was an Observer. Maybe using something like ObservingUser instead of just User would have been more intuitive. Even Subject and Blog seem somewhat strange.

None the less, the tutorial demonstrates this clearly but I think it could use some improvement in making it more intuitive. Keep up the good work. Thanks Joe.

Reply

Md Farooq *12/08/2014 at 7:13 am*

Nice article

Reply

Your Comment

Name

Email

Post Comment

TUTORIAL MENU

- Java
- Android
- Design Patterns
- Creational Design Patterns

Servlet

Site Map