ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Observer Design Pattern

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Observer Pattern

Sports Lobby is a fantastic sports site for sport lovers. They cover almost all kinds of sports and provide the latest news, information, matches scheduled dates, information about a particular player or a team. Now, they are planning to provide live commentary or scores of matches as an SMS service, but only for their premium users. Their aim is to SMS the live score, match situation, and important events after short intervals. As a user, you need to subscribe to the package and when there is a live match you will get an SMS to the live commentary. The site also provides an option to unsubscribe from the package whenever you want to.

As a developer, the Sport Lobby asked you to provide this new feature for them. The reporters of the Sport Lobby will sit in the commentary box in the match, and they will update live commentary to a commentary object. As a developer your job is to provide the commentary to the registered users by fetching it from the commentary object when it's available. When there is an update, the system should update the subscribed users by sending them the SMS.

This situation clearly shows one-to-many mapping between the match and the users, as there could be many users to subscribe to a single match. The Observer Design Pattern is best suited to this situation, let's see about this pattern and then create the feature for Sport Lobby.

2. What is the Observer Pattern

The Observer Pattern is a kind of behavior pattern which is concerned with the assignment of responsibilities between objects. The behavior patterns characterize complex control flows that are difficult to follow at run-time. They shift your focus away from the flow of control to let you concentrate just on the way objects are interconnected.

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The Observer pattern describes these dependencies. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in its state. In response, each observer will query the subject to synchronize its state with the subject state.

The other way to understand the Observer Pattern is the way Publisher-Subscriber relationship works. Let's assume for example that you subscribe to a magazine for your favorite sports or fashion magazine. Whenever a new issue is published, it gets delivered to you. If you unsubscribe from it when you don't want the magazine anymore, it will not get delivered to you. But the publisher continues to work as before, since there are other people who are also subscribed to that particular magazine.

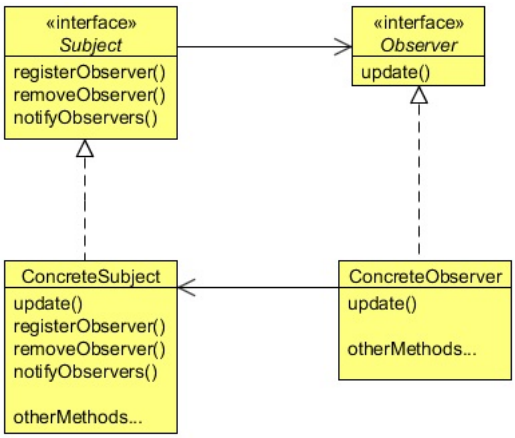

Figure 1

There are four participants in the Observer pattern:

1. Subject, which is used to register observers. Objects use this interface to register as observers and also to remove themselves from being observers.
2. Observer, defines an updating interface for objects that should be notified of changes in a subject. All observers need to implement the Observer interface. This interface has a method

   ```
   update()
   ```

   , which gets called when the Subject's state changes.
3. ConcreteSubject, stores the state of interest to ConcreteObserver objects. It sends a notification to its observers when its state changes. A concrete subject always implements the Subject interface. The

   ```
   notifyObservers()
   ```

   method is used to update all the current observers whenever the state changes.
4. ConcreateObserver, maintains a reference to a ConcreteSubject object and implements the Observer interface. Each observer registers with a concrete subject to receive updates.

3. Implementing Observer Pattern

Let's see how to use the Observer Pattern in developing the feature for the Sport Lobby. Someone will update the concrete subject's object and your job is to update the state of the object registered with the concrete subject object. So, whenever there is a change in the state of the concrete subject's object all its dependent objects should get notified and then updated.

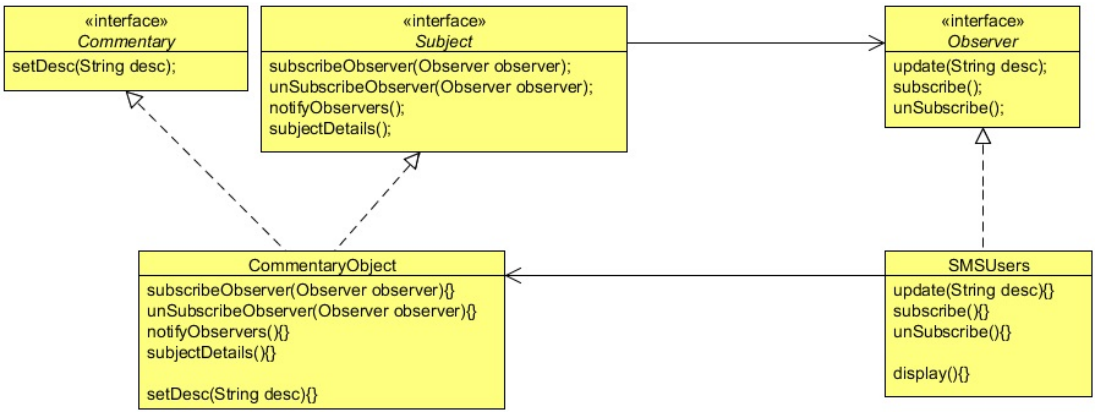

Figure 2

Based on this, let's first create a

`Subject`

interface. In the Subject interface there are three key methods and optionally, if required, you can add some more methods according to your need.

```
1  package com.javacodegeeks.patterns.observerpattern;
2
3  public interface Subject {
4
5      public void subscribeObserver(Observer observer);
6      public void unSubscribeObserver(Observer observer);
7      public void notifyObservers();
8      public String subjectDetails();
9  }
```

The three key methods in the

`Subject`

interface are:

`subscribeObserver`

, which is used to subscribe observers or we can say register the observers so that if there is a change in the state of the subject, all these observers should get notified.

`unSubscribeObserver`

, which is used to unsubscribe observers so that if there is a change in the state of the subject, this unsubscribed observer should not get notified.

`notifyObservers`

, this method notifies the registered observers when there is a change in the state of the subject.

And optionally there is one more method

`subjectDetails()`

, it is a trivial method and is according to your need. Here, its job is to return the details of the subject.

Now, let's see the

`Observer`

interface.

```
1  package com.javacodegeeks.patterns.observerpattern;
2
3  public interface Observer {
4
5      public void update(String desc);
6      public void subscribe();
7      public void unSubscribe();
8  }
```

`update(String desc)`

, method is called by the subject on the observer in order to notify it, when there is a change in the state of the subject.

`subscribe()`

, method is used to subscribe itself with the subject.

```
unsubscribe()
```

, method is used to unsubscribe itself with the subject.

```
1  package com.javacodegeeks.patterns.observerpattern;
2
3  public interface Commentary {
4
5      public void setDesc(String desc);
6  }
```

The above interface is used by the reporters to update the live commentary on the commentary object. It's an optional interface just to follow the **code to interface principle**, not related to the Observer pattern. You should apply oops principles along with the design patterns wherever applicable. The interface contains only one method which is used to change the state of the concrete subject object.

```
01 package com.javacodegeeks.patterns.observerpattern;
02
03 import java.util.List;
04
05 public class CommentaryObject implements Subject,Commentary{
06
07     private final List<Observer>observers;
08     private String desc;
09     private final String subjectDetails;
10
11     public CommentaryObject(List<Observer>observers,String subjectDetails){
12         this.observers = observers;
13         this.subjectDetails = subjectDetails;
14     }
15     @Override
16     public void subscribeObserver(Observer observer) {
17         observers.add(observer);
18     }
19
20     @Override
21     public void unSubscribeObserver(Observer observer) {
22         int index = observers.indexOf(observer);
23         observers.remove(index);
24
25     }
26
27     @Override
28     public void notifyObservers() {
29         System.out.println();
30         for(Observer observer : observers){
31             observer.update(desc);
32         }
33
34     }
35
36     @Override
37     public void setDesc(String desc) {
38         this.desc = desc;
39         notifyObservers();
40     }
41     @Override
42     public String subjectDetails() {
43         return subjectDetails;
44     }
45
46 }
```

The above class works as a concrete subject which implements the Subject interface and provides implementation of it. It also stores the reference to the observers registered to it.

```
01 package com.javacodegeeks.patterns.observerpattern;
02
03 public class SMSUsers implements Observer{
04
05     private final Subject subject;
06     private String desc;
07     private String userInfo;
08
09     public SMSUsers(Subject subject,String userInfo){
10         if(subject==null){
11             throw new IllegalArgumentException("No Publisher found.");
12         }
13         this.subject = subject;
14         this.userInfo = userInfo;
15     }
16
17     @Override
18     public void update(String desc) {
19         this.desc = desc;
20         display();
21     }
22
23     private void display(){
24         System.out.println("["+userInfo+"]: "+desc);
25     }
26
27     @Override
28     public void subscribe() {
29         System.out.println("Subscribing "+userInfo+" to "+subject.subjectDetails()+" ...");
30         this.subject.subscribeObserver(this);
31         System.out.println("Subscribed successfully.");
32     }
```

```
33
34      @Override
35      public void unSubscribe() {
36          System.out.println("Unsubscribing "+userInfo+" to "+subject.subjectDetails()+" ...");
37          this.subject.unSubscribeObserver(this);
38          System.out.println("Unsubscribed successfully.");
39      }
40
41  }
```

The above class is the concrete observer class which implements the

```
Observer
```

interface. It also stores the reference to the subject it subscribed and optionally a

```
userInfo
```

variable which is used to display the user information.

Now, let's test the example.

```
01  package com.javacodegeeks.patterns.observerpattern;
02
03  import java.util.ArrayList;
04
05  public class TestObserver {
06
07      public static void main(String[] args) {
08          Subject subject = new CommentaryObject(new ArrayList<Observer>(), "Soccer Match [2014AUG24]");
09          Observer observer = new SMSUsers(subject, "Adam Warner [New York]");
10          observer.subscribe();
11
12          System.out.println();
13
14          Observer observer2 = new SMSUsers(subject, "Tim Ronney [London]");
15          observer2.subscribe();
16
17          Commentary cObject = ((Commentary)subject);
18          cObject.setDesc("Welcome to live Soccer match");
19          cObject.setDesc("Current score 0-0");
20
21          System.out.println();
22
23          observer2.unSubscribe();
24
25          System.out.println();
26
27          cObject.setDesc("It's a goal!!");
28          cObject.setDesc("Current score 1-0");
29
30          System.out.println();
31
32          Observer observer3 = new SMSUsers(subject, "Marrie [Paris]");
33          observer3.subscribe();
34
35          System.out.println();
36
37          cObject.setDesc("It's another goal!!");
38          cObject.setDesc("Half-time score 2-0");
39
40      }
41
42  }
```

The above example will produce the following output:

```
01  Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
02  Subscribed successfully.
03
04  Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
05  Subscribed successfully.
06
07  [Adam Warner [New York]]: Welcome to live Soccer match
08  [Tim Ronney [London]]: Welcome to live Soccer match
09
10  [Adam Warner [New York]]: Current score 0-0
11  [Tim Ronney [London]]: Current score 0-0
12
13  Unsubscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
14  Unsubscribed successfully.
15
16  [Adam Warner [New York]]: It's a goal!!
17
18  [Adam Warner [New York]]: Current score 1-0
19
20  Subscribing Marrie [Paris] to Soccer Match [2014AUG24] ...
21  Subscribed successfully.
22
23  [Adam Warner [New York]]: It's another goal!!
24  [Marrie [Paris]]: It's another goal!!
25
26  [Adam Warner [New York]]: Half-time score 2-0
27  [Marrie [Paris]]: Half-time score 2-0
```

As you can see, at first two users subscribed themselves for the soccer match and started receiving the commentary. But later one user unsubscribed it, so the user did not receive the commentary again. Then, another user subscribed and starts getting the commentary.

All this happens dynamically without changing the existing code and not only this, suppose if, the company wants to broadcast the commentary on emails or any other firm wants to do collaboration with this company to broadcast the commentary. All you need to do is to create two new classes like

```
UserEmail
```

and

```
ColCompany
```

and make them observer of the subject by implementing the

```
Observer
```

interface. As far as the

```
Subject
```

knows it's an observer, it will provide the update.

4. Java's built-in Observer Pattern

Java has built-in support for the

```
Observer
```

Pattern. The most general is the Observer interface and the Observable class in the

```
java.util
```

package. These are quite similar to our Subject and Observer interface, but give you a lot of functionality out of the box.

Let's try to implement the above example using the Java's built-in Observer pattern.

```
01 package com.javacodegeeks.patterns.observerpattern;
02
03 import java.util.Observable;
04
05 public class CommentaryObjectObservable extends Observable implements Commentary {
06     private String desc;
07     private final String subjectDetails;
08
09     public CommentaryObjectObservable(String subjectDetails){
10         this.subjectDetails = subjectDetails;
11     }
12
13     @Override
14     public void setDesc(String desc) {
15         this.desc = desc;
16         setChanged();
17         notifyObservers(desc);
18     }
19
20     public String subjectDetails() {
21         return subjectDetails;
22     }
23 }
```

This time we extends the

```
Observable
```

class to make our class as a subject and please note that the above class does not hold any reference to the observers, it is handled by the parent class, that's is, the

```
Observable
```

class. However, we declared the

```
setDesc
```

method to change the state of the object, as done in the previous example. The

```
setChanged
```

method is the method from the upper class which is used to set the changed flag to true. The

```
notifyObservers
```

method notifies all of its observers and then calls the

```
clearChanged
```

method to indicate that this object has no longer changed. Each observer has its update method called with two arguments: an

```
observable
```

object and the

```
arg
```

argument.

```
01 package com.javacodegeeks.patterns.observerpattern;
02
03 import java.util.Observable;
04
05 public class SMSUsersObserver implements java.util.Observer{
06
07     private String desc;
08     private final String userInfo;
09     private final Observable observable;
10
11     public SMSUsersObserver(Observable observable,String userInfo){
12         this.observable = observable;
13         this.userInfo = userInfo;
14     }
15
16     public void subscribe() {
17         System.out.println("Subscribing "+userInfo+" to "+((CommentaryObjectObservable)
(observable)).subjectDetails()+" ...");
18         this.observable.addObserver(this);
19         System.out.println("Subscribed successfully.");
20     }
21
22     public void unSubscribe() {
23         System.out.println("Unsubscribing "+userInfo+" to "+((CommentaryObjectObservable)
(observable)).subjectDetails()+" ...");
24         this.observable.deleteObserver(this);
25         System.out.println("Unsubscribed successfully.");
26     }
27
28     @Override
29     public void update(Observable o, Object arg) {
30             desc = (String)arg;
31             display();
32     }
33
34     private void display(){
35         System.out.println("["+userInfo+"]: "+desc);
36     }
37
38 }
```

Let's discuss some of the key methods.

The above class implements the

```
Observer
```

interface which has one key method

```
update
```

, which is called when the subject calls the

```
notifyObservers
```

method. The

```
update
```

method takes an

```
Observable
```

object and an

```
arg
```

as parameters.

The

```
addObserver
```

method is used to register a observer to the subject, and the

```
deleteObserver
```

method is used to remove the observer from the subject's list.

Let's test this example.

```
01 package com.javacodegeeks.patterns.observerpattern;
02
03 public class Test {
04
05     public static void main(String[] args) {
06         CommentaryObjectObservable obj = new CommentaryObjectObservable("Soccer Match [2014AUG24]");
07         SMSUsersObserver observer = new SMSUsersObserver(obj, "Adam Warner [New York]");
08         SMSUsersObserver observer2 = new SMSUsersObserver(obj,"Tim Ronney [London]");
09         observer.subscribe();
10         observer2.subscribe();
11         System.out.println("-------------------------------------------------------");
```

```
        obj.setDesc("Welcome to live Soccer match");
        obj.setDesc("Current score 0-0");

        observer.unSubscribe();

        obj.setDesc("It's a goal!!");
        obj.setDesc("Current score 1-0");
    }
}
```

The above example will produce the following output:

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Subscribed successfully.
Subscribing Tim Ronney [London] to Soccer Match [2014AUG24] ...
Subscribed successfully.
--------------------------------------------------
[Tim Ronney [London]]: Welcome to live Soccer match
[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Current score 0-0
[Adam Warner [New York]]: Current score 0-0
Unsubscribing Adam Warner [New York] to Soccer Match [2014AUG24] ...
Unsubscribed successfully.
[Tim Ronney [London]]: It's a goal!!
[Tim Ronney [London]]: Current score 1-0
```

The above class creates a subject and two observers. The

`subscribe`

method of the

`observer`

adds itself to the subject observers list. Then

`setDesc`

changes the state of the subject which call the

`setChanged`

method to set the change flag to true, and notifies the observers. As a result, observer's

`update`

method is called which internally class the

`display`

method to display the result. Later, one of the observers gets

`unsubscribe`

d, i.e. it is removed from the observer's list. Due to which later commentaries were not updated to it.

Java provides built-in facility for the Observer Pattern, but it comes with its own drawbacks. The

`Observable`

is a class, you have to subclass it. That means you can't add on the Observable behavior to an existing class that already extends another superclass. This limits the reuse potential. You can't even create your own implementation that plays well with Java's built-in Observer API. Some of the methods in the Observable API are protected. This means you can't call the methods like

`setChange`

unless you've subclassed

`Observable`

. And you can't even create an instance of the

`Observable`

class and compose it with your own objects, you have to subclass. This design violates the **"favor composition over inheritance"** design principle.

5. When to use the Observer Pattern

Use the Observer pattern in any of the following situations:

1. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
2. When a change to one object requires changing others, and you don't know how many objects need to be changed.
3. When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.