Observer Design Pattern

Last Modified: August 23, 2021

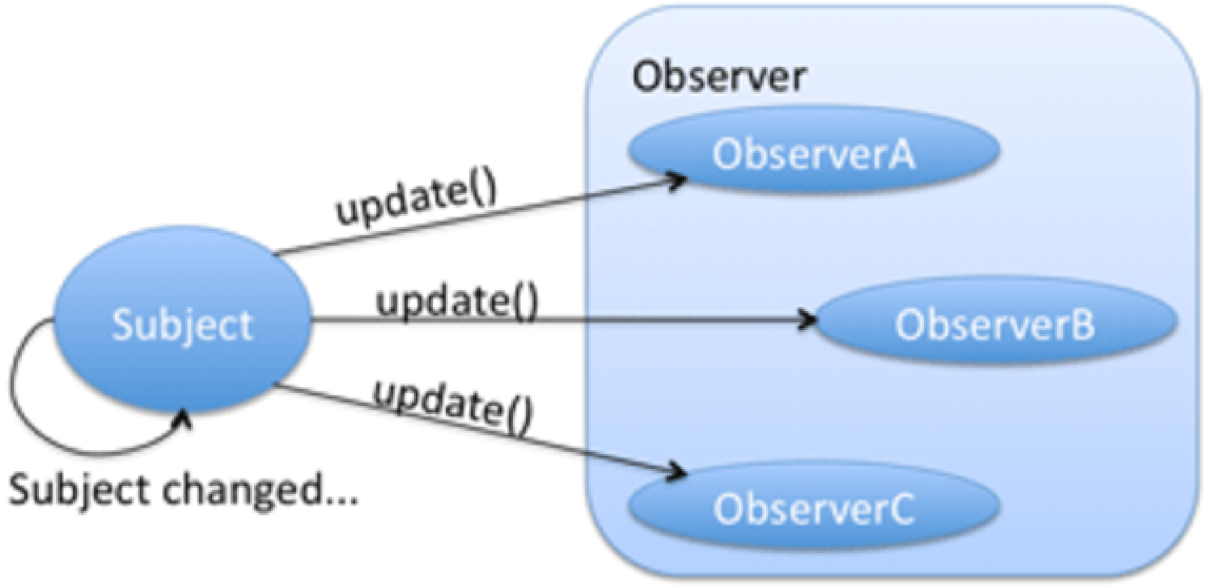
According to GoF definition, **observer pattern** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the **publish-subscribe pattern**.

In observer pattern, there are many observers (subscriber objects) that are observing a particular subject (publisher object). Observers register themselves to a subject to get a notification when there is a change made inside that subject.

A observer object can register or unregister from subject at any point of time. It helps is making the objects objects **loosely coupled**.

1. When to use observer design pattern

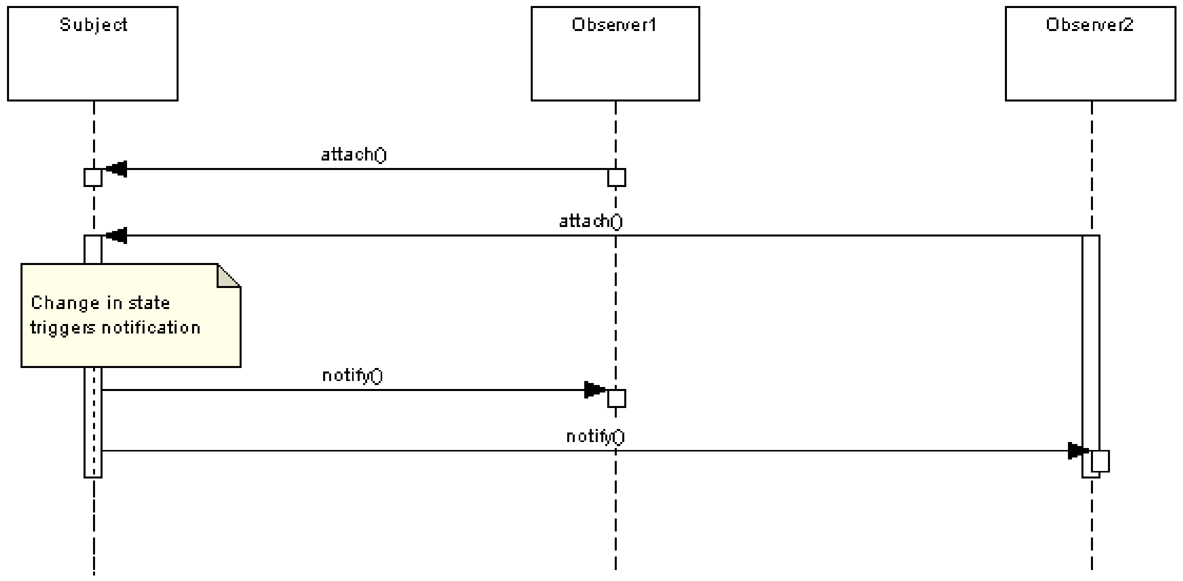
As described above, when you have a design a system where multiple entities are interested in any possible update to some particular second entity object, we can use the observer pattern.



Observer Pattern

The flow is very simple to understand. Application creates the concrete subject object. All concrete observers register themselves to be notified for any further update in the state of subject.

As soon as the state of subject changes, subject notifies all the registered observers and the observers can access the updated state and act accordingly.



Observer Pattern Sequence Diagram

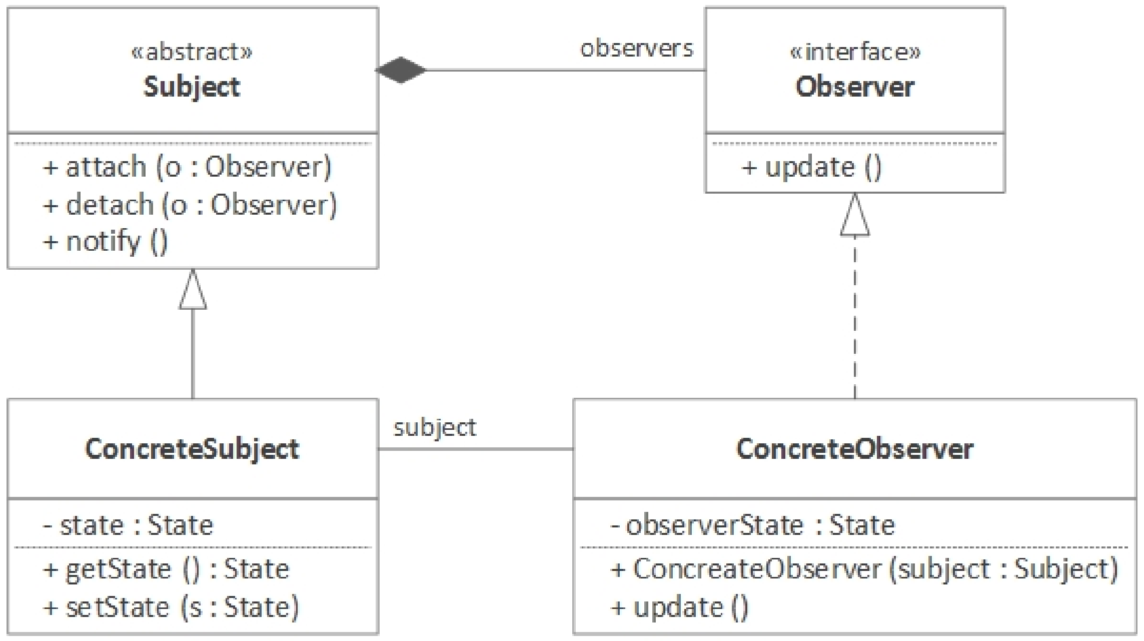
2. Real world example of observer pattern

- A real world example of observer pattern can be any social media platform such as Facebook or twitter. When a person updates his status – all his followers gets the notification.
  A follower can follow or unfollow another person at any point of time. Once unfollowed, person will not get the notifications from subject in future.

- In programming, observer pattern is the basis of message oriented applications. When a application has updated it's state, it notifies the subscribers about updates. Frameworks like HornetQ, JMS work on this pattern.
- Similarly, Java UI based programming, all keyboard and mouse events are handled by it's listeners objects and designated functions. When user click the mouse, function subscribed to the mouse click event is invoked with all the context data passed to it as method argument.

3. Observer design pattern

3.1. Architecture



Observer Pattern Architecture

3.2. Design participants

The observer pattern has four participants.

- **Subject** – interface or abstract class defining the operations for attaching and de-attaching observers to the subject.
- **ConcreteSubject** – concrete Subject class. It maintain the state of the object and when a change in the state occurs it notifies the attached Observers.

- **Observer** – interface or abstract class defining the operations to be used to notify this object.
- **ConcreteObserver** – concrete Observer implementations.

4. Observer design pattern example

In below example, I am creating a message publisher of type `Subject` and three subscribers of type `Observer`. Publisher will publish the message periodically to all subscribed or attached observers and they will print the updated message to console.

Subject and ConcreteSubject

Subject.java

```
public interface Subject
{
    public void attach(Observer o);
    public void detach(Observer o);
    public void notifyUpdate(Message m);
}
```

MessagePublisher.java

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

public class MessagePublisher implements Subject {

    private List<Observer> observers = new ArrayList<>();
    @Override
    public void attach(Observer o) {
        observers.add(o);
    }

    @Override
    public void detach(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyUpdate(Message m) {
        for(Observer o: observers) {
            o.update(m);
        }
    }
}
```

Observer and ConcreteObservers

Observer.java

```
public interface Observer
{
    public void update(Message m);
}
```

MessageSubscriberOne.java

```
public class MessageSubscriberOne implements Observer
```

```
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberOne :: " + m.getMessageContent());
    }
}
```

MessageSubscriberTwo.java

```
public class MessageSubscriberTwo implements Observer
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberTwo :: " + m.getMessageContent());
    }
}
```

MessageSubscriberThree.java

```
public class MessageSubscriberThree implements Observer
{
    @Override
    public void update(Message m) {
        System.out.println("MessageSubscriberThree :: " + m.getMessageContent());
    }
}
```

State object

This must be an immutable object so that no class can modify it's content by mistake.

Message.java

```
public class Message
{
    final String messageContent;

    public Message (String m) {
        this.messageContent = m;
    }

    public String getMessageContent() {
        return messageContent;
    }
}
```

Now test the communication between publisher and subscribers.

Main.java

```
public class Main
{
    public static void main(String[] args)
    {
        MessageSubscriberOne s1 = new MessageSubscriberOne();
        MessageSubscriberTwo s2 = new MessageSubscriberTwo();
        MessageSubscriberThree s3 = new MessageSubscriberThree();

        MessagePublisher p = new MessagePublisher();
        p.attach(s1);
        p.attach(s2);

        p.notifyUpdate(new Message("First Message"));   //s1 and s2 will receive the update

        p.detach(s1);
        p.attach(s3);

        p.notifyUpdate(new Message("Second Message")); //s2 and s3 will receive the update
    }
}
```

Program output.

```
Console

MessageSubscriberOne :: First Message
MessageSubscriberTwo :: First Message

MessageSubscriberTwo :: Second Message
MessageSubscriberThree :: Second Message
```

5. FAQs

- **Can different types of observers register to one subject?**
  The nature and functionality of observers can be different but they all must implement the one common `Observer` interface which the subject support for registering and deregistering.

- **Can I add or remove observers at runtime?**

Yes. We can add or remove the observers at any point of time.

- **Difference between observer pattern and chain of responsibility pattern?**
  In an observer pattern, all registered handler objects get notifications at the same time and they process the update at same time.

  But in a chain of responsibility pattern, handler objects in the chain are notified one by one, and this process continues until one object fully handles the notification.

- **Benefits of the observer pattern?**
  The subject and observers make a loosely coupled system. They do not need to know each other explicitly. We can independently add or remove observers at any time.

Related Java classes:

Observer Java Doc
Observable Java Doc

Share this:

Twitter Facebook LinkedIn Reddit WhatsApp

Subscribe to Blog

Enter your email address to subscribe and receive notifications of new posts by email.

Join 3,810 other subscribers

Email Address

SUBSCRIBE

About Lokesh Gupta

A family guy with fun loving nature. Love computers, programming and solving everyday problems. Find me on Facebook and Twitter.

Leave a Reply

Enter your comment here...

Search Tutorials

Type and Press ENTER...

Meta Links

About Me

Contact Us

Privacy policy

Advertise

Recommended Reading

10 Life Lessons

Secure Hash Algorithms

How Web Servers work?

How Java I/O Works Internally?

Guest and Sponsored Posts

Best Way to Learn Java

Java Best Practices Guide

Microservices Tutorial

REST API Tutorial

How to Start New Blog