Understanding software design patterns

Design patterns help eliminate redundant coding. Learn how to use the singleton pattern, factory pattern, and observer pattern using Java.

By Bryant Son (Alumni)

July 15, 2019 | 4 Comments | 10 min read

Register or Login to like

Image by: *Opensource.com*

If you are a programmer or a student pursuing computer science or a similar discipline, sooner or later, you will encounter the term "software design pattern." According to Wikipedia, *"a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design."* Here is my take on the definition: When you have been working on a coding project for a while, you often begin to think, "Huh, this seems redundant. I wonder if I can change the code to be more flexible and accepting of changes?" So, you begin to think about how to separate what stays the same from what needs to change often.

> A **design pattern** is a way to make your code easier to change by separating the part that stays the same and the part that needs constant changes.

Not surprisingly, everyone who has worked on a programming project has probably had the same thought. Especially for any industry-level project, where it's common to work with dozens or even hundreds of developers; the collaboration process suggests that there have to be some standards and rules to make the code more elegant and adaptable to changes. That is why we have object-oriented programming (OOP) and software framework tools. A design pattern is somewhat similar to OOP, but it goes further by considering changes as part of the natural development process. Basically, the design pattern leverages some ideas from OOP, like abstractions and interfaces, but focuses on the process of changes.

When you start to work on a project, you often hear the term *refactoring*, which means *to change the code to be more elegant and reusable;* this is where the design pattern shines. Whenever you're working on existing code (whether built by someone else or your past self), knowing the design patterns helps you begin to see things differently—you will discover problems and ways to improve the code.

There are numerous design patterns, but three popular ones, which I'll present in this introductory article, are singleton pattern, factory pattern, and observer pattern.

How to follow this guide

I want this tutorial to be as easy as possible for anyone to understand, whether you are an experienced programmer or a beginner to coding. The design pattern concept is not exactly easy to understand, and reducing the learning curve when you start a journey is always a top priority. Therefore, in addition to this article with diagrams and code pieces, I've also created a GitHub repository you can clone and run the code to implement the three design patterns on your own. You can also follow along with the following YouTube video I created.

Prerequisites

If you just want to get the idea of design patterns in general, you do not need to clone the sample project or install any of the tools. However, to run the sample code, you need to have the following installed:

> **Java Development Kit (JDK):** I highly recommend OpenJDK.
>
> **Apache Maven:** The sample project is built using Apache Maven; fortunately, many IDEs come with Maven installed.
>
> **Interactive development editor (IDE):** I use IntelliJ Community Edition, but you can use Eclipse IDE or any other Java IDE of your choice
>
> **Git:** If you want to clone the project, you need a Git client.

To clone the project and follow along, run the following command after you install Git:

```
git clone https://github.com/bryantson/OpensourceDotComDemos.git
```

Then, in your favorite IDE, you can import the code in the TopDesignPatterns repo as an Apache Maven project.

I am using Java, but you can implement the design pattern using any programming language that supports the abstraction principle.

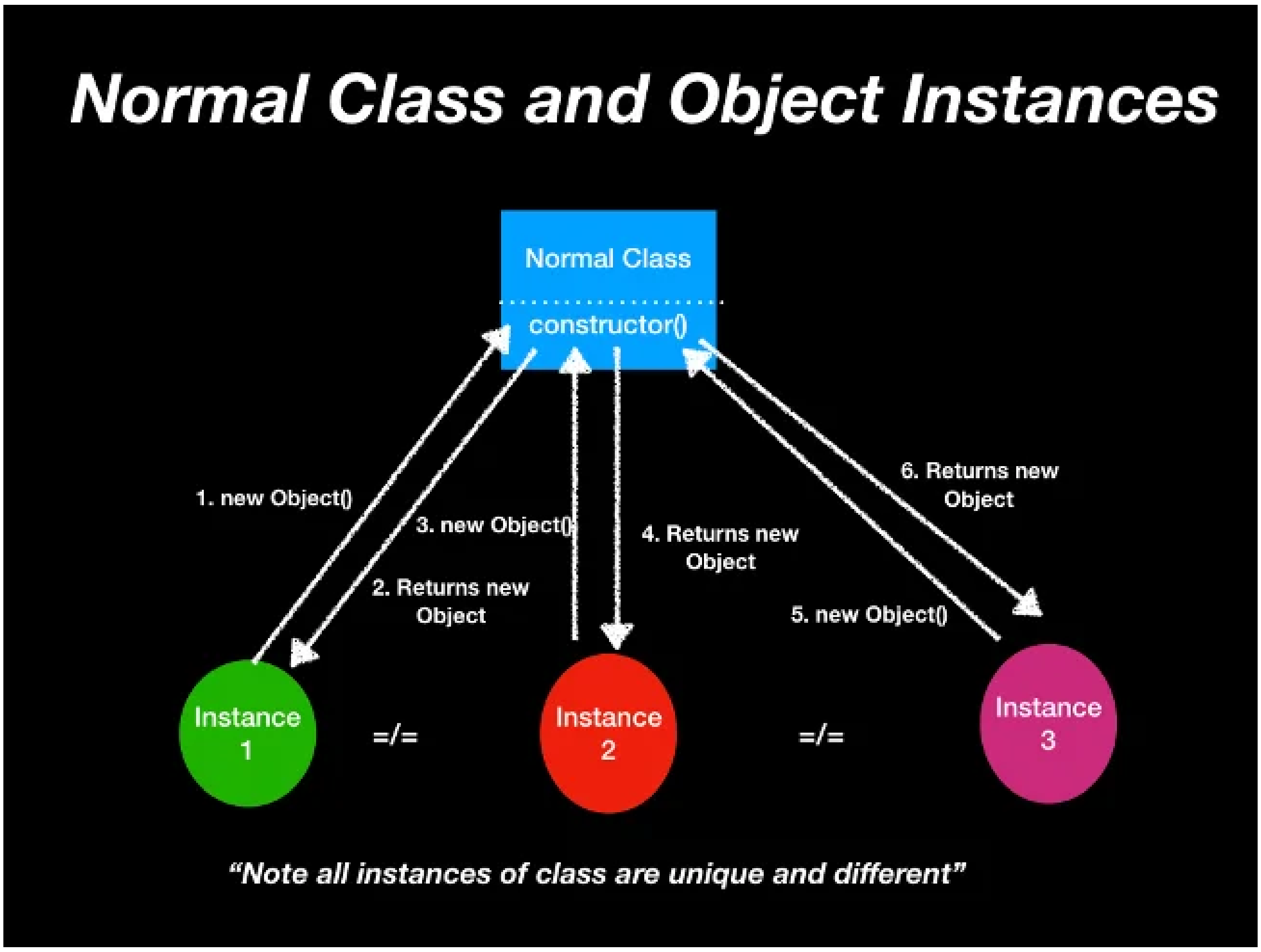
Singleton pattern: Avoid creating an object every single time

The singleton pattern is a very popular design pattern that is also relatively simple to implement because you need just one class. However, many developers debate whether the singleton design pattern's benefits outpace its problems because it lacks clear benefits and is easy to abuse. Few developers implement singleton directly; instead, programming frameworks like Spring Framework and Google Guice have built-in singleton design pattern features.

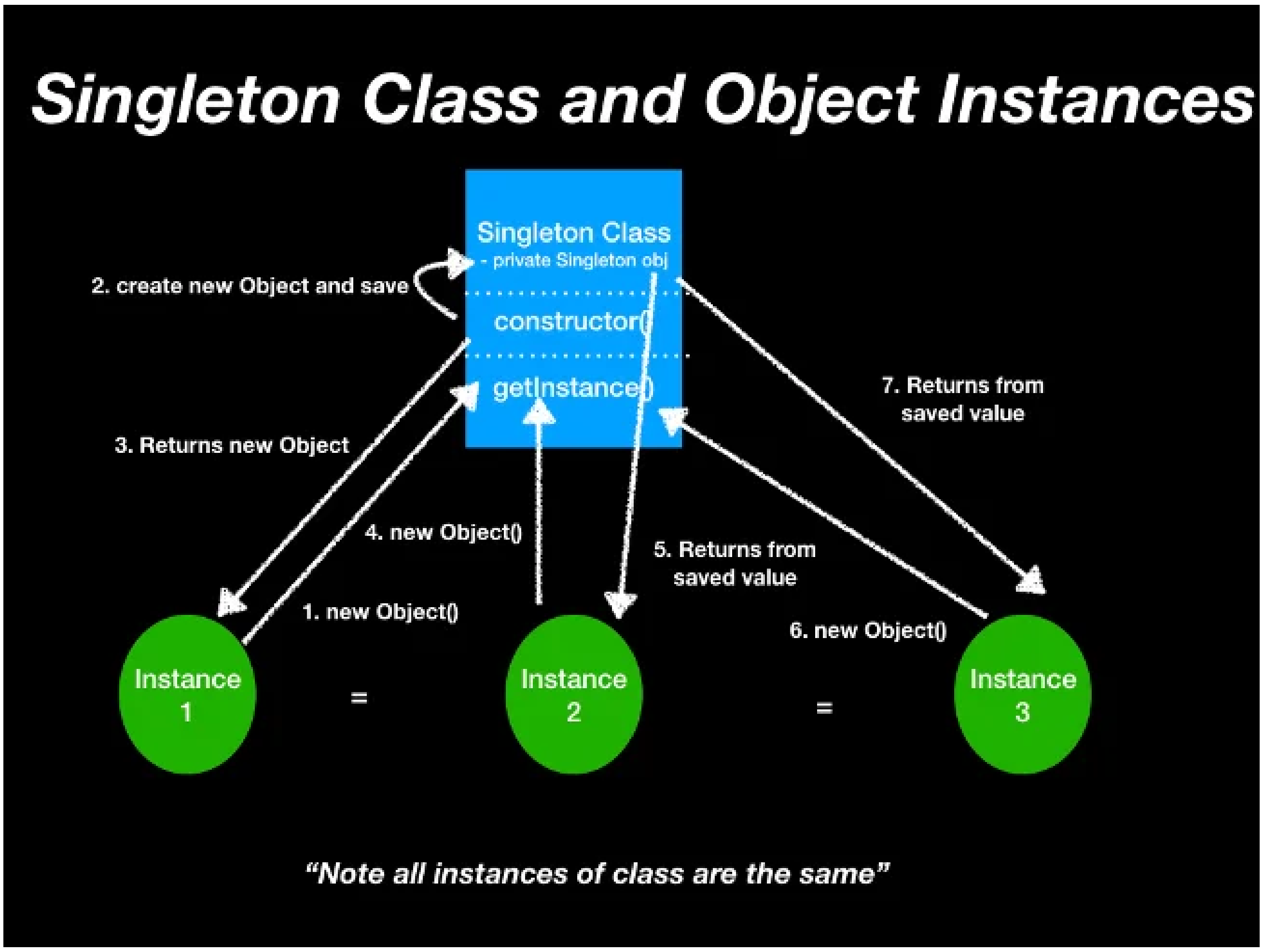
But knowing about singleton is still tremendously useful. The singleton pattern makes sure that a class is created only once and provides a global point of access to it.

> **Singleton pattern:** Ensures that only one instantation is created and avoids creating multiple instances of the same object.

The diagram below shows the typical process for creating a class object. When the client asks to create an object, the constructor creates, or instantiates, an object and returns to the class with the caller method. However, this happens every single time an object is requested—the constructor is called, a new object is created, and it returns with a unique object. I guess the creators of the OOP language had a reason behind creating a new object every single time, but the proponents of the singleton process say this is redundant and a waste of resources.



The following diagram creates the object using the singleton pattern. Here, the constructor is called only when the object is requested the first time through a designated getInstance() method. This is usually done by checking the null value, and the object is saved inside the singleton class as a private field value. The next time the getInstance() is called, the class returns the object that was created the first time. No new object is created; it just returns the old one.



The following script shows the simplest possible way to create the singleton pattern:

```java
package org.opensource.demo.singleton;

public class OpensourceSingleton {

    private static OpensourceSingleton uniqueInstance;

    private OpensourceSingleton() {
    }

    public static OpensourceSingleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new OpensourceSingleton();
        }
        return uniqueInstance;
    }

}
```

On the caller side, here is how the singleton class will be called to get an object:

```
Opensource newObject = Opensource.getInstance();
```

This code demonstrates the idea of a singleton well:

1. When getInstance() is called, it checks whether the object was already created by checking the null value.
2. If the value is null, it creates a new object, saves it into the private field, and returns the object to the caller. Otherwise, it returns the object that was created previously.

The main problem with this singleton implementation is its disregard for parallel processes. When multiple processes using threads access the resource simultaneously, a problem occurs. There is one solution to this, and it is called *double-checked locking* for multithread safety, which is shown here:

```
package org.opensource.demo.singleton;

public class ImprovedOpensourceSingleton {

    private volatile static ImprovedOpensourceSingleton uniqueInst

    private ImprovedOpensourceSingleton() {}

    public static ImprovedOpensourceSingleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (ImprovedOpensourceSingleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new ImprovedOpensourceSinglet
                }
            }
        }
        return uniqueInstance;
    }

}
```

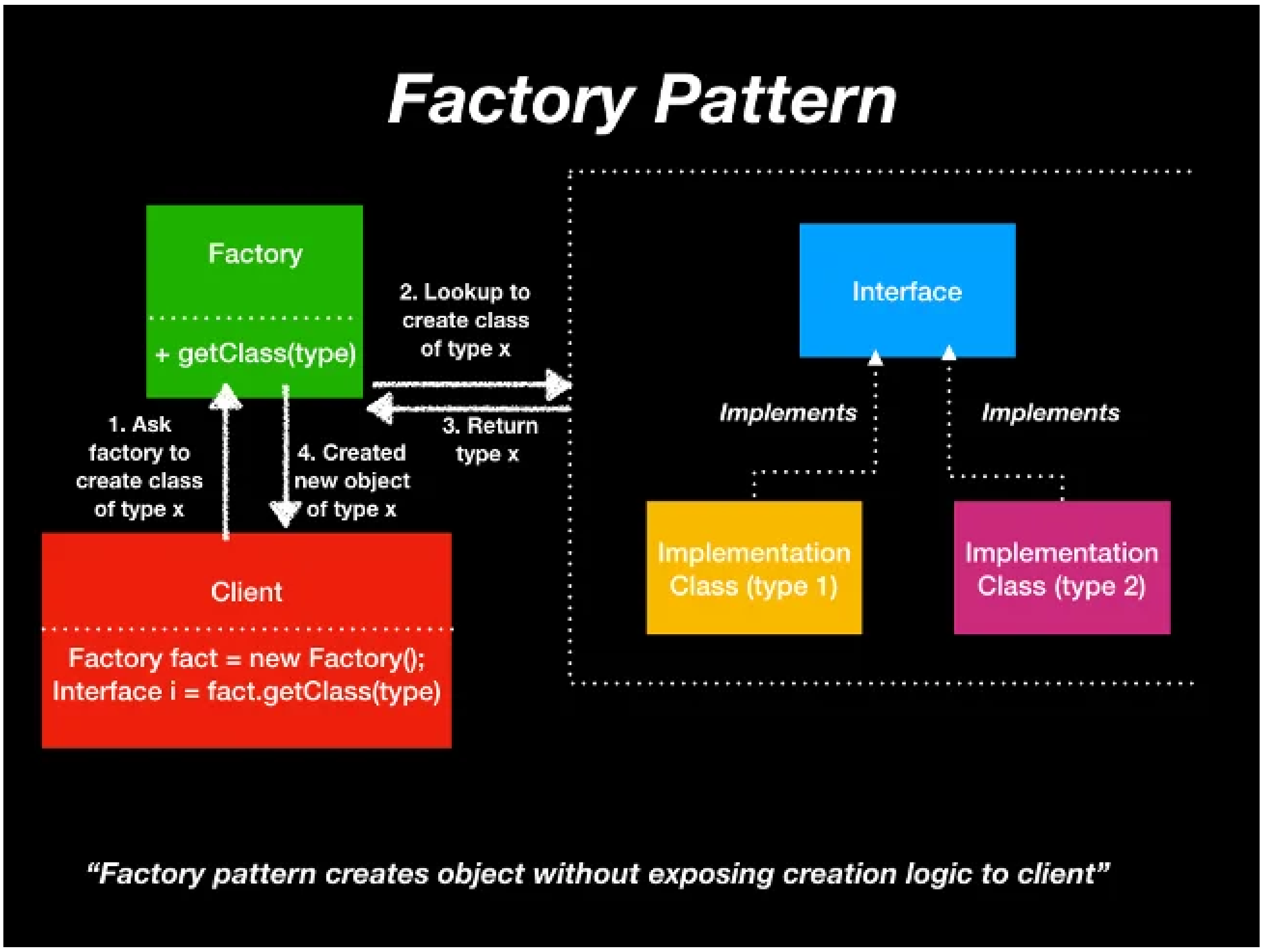
Just to emphasize the previous point, make sure to implement your singleton directly only when you believe is a safe option to do so. The best way is to leverage the singleton feature is by using a well-made programming framework.

Factory pattern: Delegate object creation to the factory class to hide creation logic

The factory pattern is another well-known design pattern, but it is a little more complex. There are several ways to implement the factory pattern, but the following sample code demonstrates the simplest possible way. The factory pattern defines an interface for creating an object but lets the subclasses decide which class to instantiate.

> **Factory pattern:** Delegates object creation to the factory class so it hides the creation logic.

The diagram below shows how the simplest factory pattern is implemented.


Factory Pattern
Factory
+ getClass(type)
2. Lookup to create class of type x
3. Return type x
1. Ask factory to create class of type x
4. Created new object of type x
Client
Factory fact = new Factory();
Interface i = fact.getClass(type)
Interface
Implements
Implements
Implementation Class (type 1)
Implementation Class (type 2)
"Factory pattern creates object without exposing creation logic to client"

Instead of the client directly calling the object creation, the client asks the factory class for a certain object, type x. Based on the type, the factory pattern decides which object to create and to return.

In this code sample, OpensourceFactory is the factory class implementation that takes the *type* from the caller and decides which object to create based on that input value:

```
package org.opensource.demo.factory;

public class OpensourceFactory {
    public OpensourceJVMServers getServerByVendor(String name) {
        if(name.equals("Apache")) {
            return new Tomcat();
        }
        else if(name.equals("Eclipse")) {
            return new Jetty();
        }
        else if (name.equals("RedHat")) {
            return new WildFly();
        }
        else {
            return null;
        }
    }
}
```

And OpenSourceJVMServer is a 100% abstraction class (or an interface class) that indicates what to implement, not how:

```
package org.opensource.demo.factory;

public interface OpensourceJVMServers {
    public void startServer();
    public void stopServer();
    public String getName();
}
```

Here is a sample implementation class for OpensourceJVMServers. When "RedHat" is passed as the type to the factory class, the WildFly server is created:

```
package org.opensource.demo.factory;

public class WildFly implements OpensourceJVMServers {
    public void startServer() {
        System.out.println("Starting WildFly Server...");
    }

    public void stopServer() {
        System.out.println("Shutting Down WildFly Server...");
    }

    public String getName() {
        return "WildFly";
    }
}
```

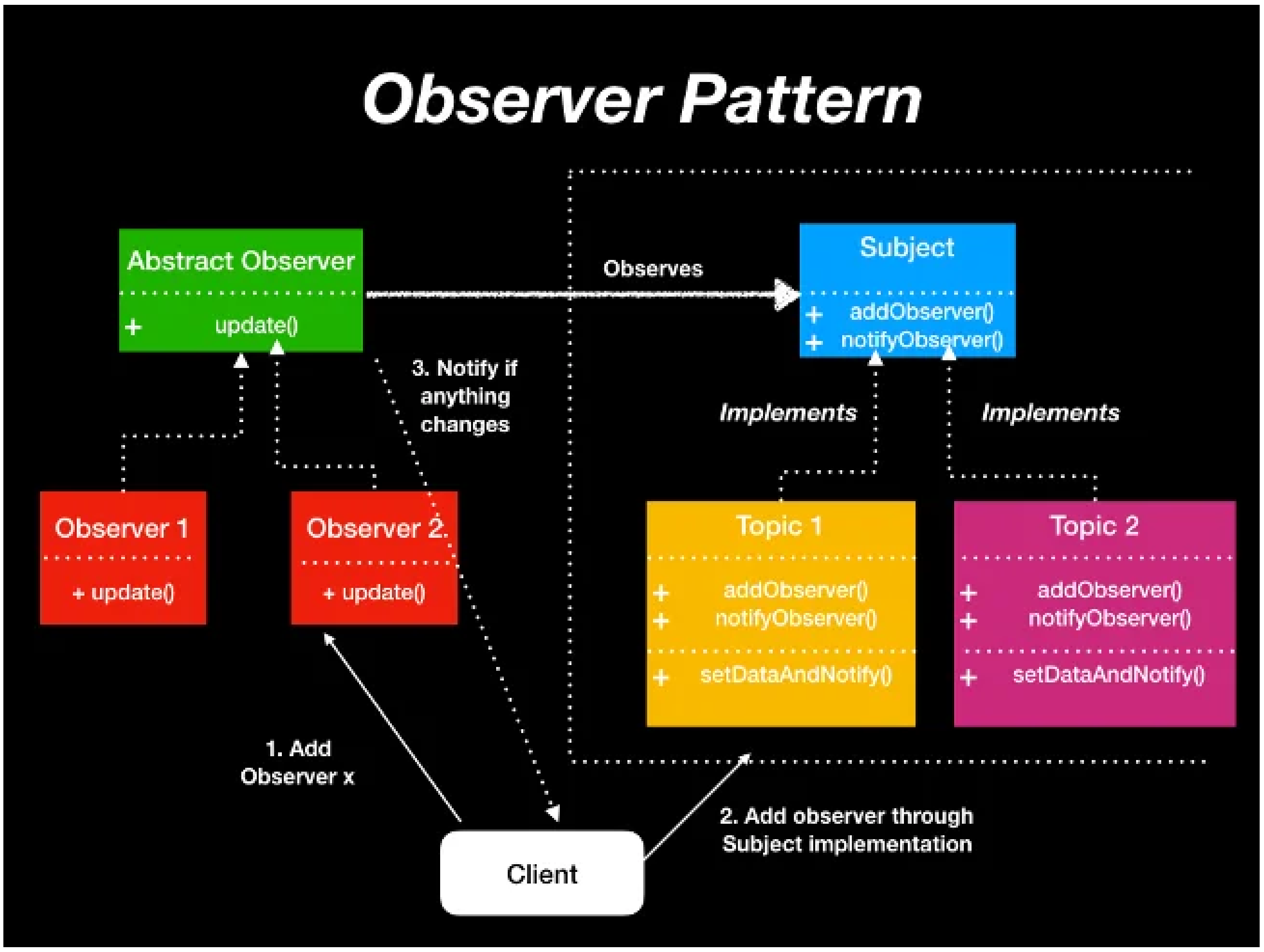
Observer pattern: Subscribe to topics and get notified about updates

Finally, there is the observer pattern. Like the singleton pattern, few professional programmers implement the observer pattern directly. However, many messaging queue and data service implementations borrow the observer pattern concept. The observer pattern defines one-to-many dependencies between objects so that when one object changes state, all of its dependents are notified and updated automatically.

> **Observer pattern:** Subscribe to the topics/subjects where the client can be notified if there is an update.

The easiest way to think about the observer pattern is to imagine a mailing list where you can subscribe to any topic, whether it is open source, technologies, celebrities, cooking, or anything else that interests you. Each topic maintains a list of its subscribers, which is equivalent to an "observer" in the observer pattern. When a topic is updated, all of its subscribers (observers) are notified of the changes. And a subscriber can always unsubscribe from a topic.

As the following diagram shows, the client can be subscribed to different topics and add the observer to be notified about new information. Because the observer listens continuously to the subject, the observer notifies the client about any change that occurs.



Let's look at the sample code for the observer pattern, starting with the subject/topic class:

```
package org.opensource.demo.observer;

public interface Topic {

    public void addObserver(Observer observer);
    public void deleteObserver(Observer observer);
    public void notifyObservers();
}
```

This code describes an interface for different topics to implement the defined methods. Notice how an observer can be added, removed, or notified.

Here is an example implementation of the topic:

```
package org.opensource.demo.observer;

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

```
public class Conference implements Topic {
    private List<Observer> listObservers;
    private int totalAttendees;
    private int totalSpeakers;
    private String nameEvent;

    public Conference() {
        listObservers = new ArrayList<Observer>();
    }

    public void addObserver(Observer observer) {
        listObservers.add(observer);
    }

    public void deleteObserver(Observer observer) {
        int i = listObservers.indexOf(observer);
        if (i >= 0) {
            listObservers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i=0, nObservers = listObservers.size(); i < nObse
            Observer observer = listObservers.get(i);
            observer.update(totalAttendees,totalSpeakers,nameEvent
        }
    }

    public void setConferenceDetails(int totalAttendees, int total
        this.totalAttendees = totalAttendees;
        this.totalSpeakers = totalSpeakers;
        this.nameEvent = nameEvent;
        notifyObservers();
    }
}
```

This class defines the implementation of a particular topic. When a change happens, this implementation is where it is invoked. Notice that this takes the number of observers, which is stored as the list, and can both notify and maintain the observers.

Here is an observer class:

```
package org.opensource.demo.observer;
```

```
public interface Observer {
    public void update(int totalAttendees, int totalSpeakers, Stri
}
```

This class defines an interface that different observers can implement to take certain actions.

For example, the observer implementation can print out the number of attendees and speakers at a conference:

```
package org.opensource.demo.observer;

public class MonitorConferenceAttendees implements Observer {
    private int totalAttendees;
    private int totalSpeakers;
    private String nameEvent;
    private Topic topic;

    public MonitorConferenceAttendees(Topic topic) {
        this.topic = topic;
        topic.addObserver(this);
    }

    public void update(int totalAttendees, int totalSpeakers, Stri
        this.totalAttendees = totalAttendees;
        this.totalSpeakers = totalSpeakers;
        this.nameEvent = nameEvent;
        printConferenceInfo();
    }

    public void printConferenceInfo() {
        System.out.println(this.nameEvent + " has " + totalSpeaker
    }
}
```

Where to go from here?

Now that you've read this introductory guide to design patterns, you should be in a good place to pursue other design patterns, such as facade, template, and decorator. There are also concurrent and distributed system design patterns like the circuit breaker pattern and the actor pattern.

However, I believe it's best to hone your skills first by implementing these design patterns in your side projects or just as practice. You can even begin to contemplate how you can apply these design patterns in your real projects. Next, I highly recommend checking out the SOLID principles of OOP. After that, you will be ready to look into the other design patterns.