An Assignment of Observer Pattern

Course Title: ***Design***

***Course Code*: SE 3109**

***Submitted By*:**

Md. Mynuddin

Roll No: ASH1825007M

Year-03, Term-01

Session: 2017-18

Submitted To:

Falguni Roy

Assistant Professor

Software Engineering, IIT

Noakhali Science and Technology University

Date Of Submission: 11th February, 2020

"Software Engineering"

NOAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

What Is The Observer Design Pattern?

An object oriented software design pattern, in which objects have a certain relationship. It keeps the objects informed if something they care about happens that means When an object changes state, all of the objects that need to know about it are notified and updated automatically.

In the Observer pattern, objects have specific roles. Observers are objects that depend on another known as Subject. This last (Subject) notifies the entire dependent Observers of any changes. In general the Subject calls a predefined method in the Observers that handles the notification and reacts appropriately.

Observer Pattern Implementation:

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class Observer and a concrete class Subject that is extending class Observer .ObserverPatternDemo, our demo class, will use Subject and concrete class object to show observer pattern in action.

Which Pattern Category It Belongs?

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependents objects are to be notified automatically. Observer pattern falls under **behavioral pattern category**.

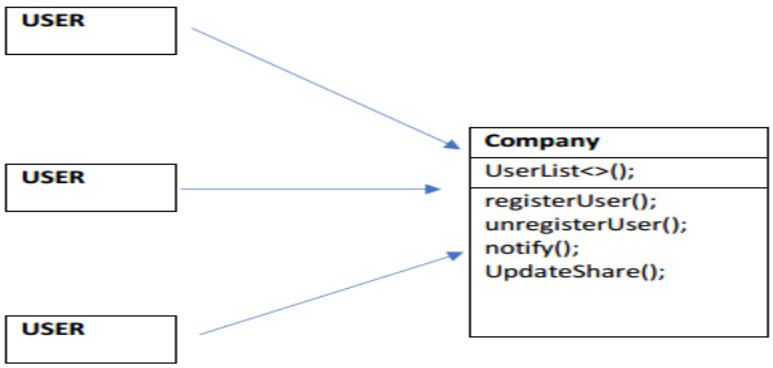
Problem (Scenario):

There are few user and company. Company has few shares and user brought those shares at a fixed price. User has to register of a company if he/she wants to buy a share of a company. The company has to keep a user list and a process. User list for keep information that are register to buy its share and process for if the share price is change then it has to give a massage to its user for inform its share price.

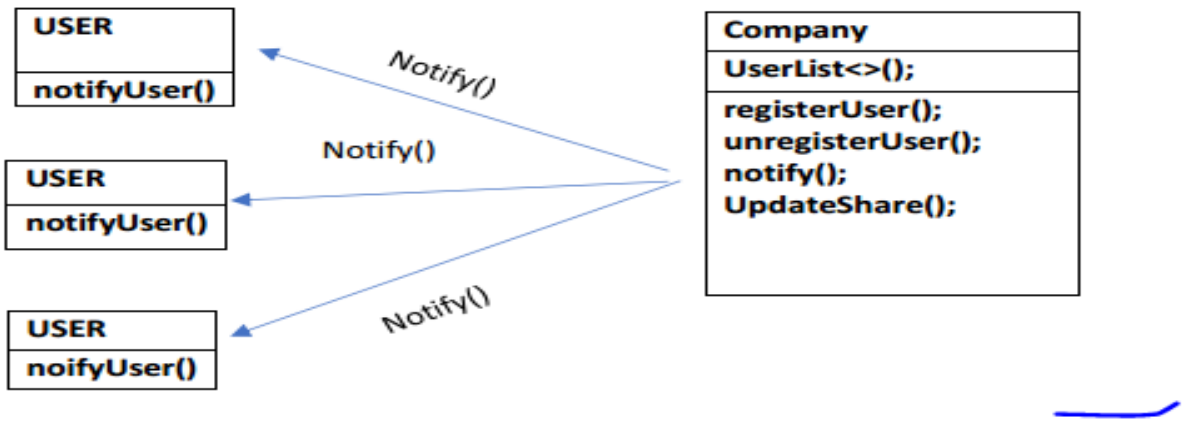
Solution:

The object that has some interesting state is often called subject, but since it's also going to notify other objects about the changes to its state, we'll call it Company. All other objects that want to track changes to the Company's state are called user.

The Observer pattern suggests that first of all we add a register mechanism to the company class so individual user can register to or unregister from a stream of events coming from that company .This mechanism consists of 1) an array field for storing a list of references to user objects and 2) several public methods which allow adding users to and removing them from that list.



Now, whenever an important event happens to the company, it goes over its user and calls the specific notification method on their objects.



Company Class contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Company class 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.

So now we will create a company class which contain two method call registerUser for add user into the userList, and UnregisterUser method for remove user from the company userList.This class also contain a changeSharePrice method which will check that current share price is equal to previous share price or not. If not then it will call another method call notifyUser.Then notifyUser method will notify all registered user of that company about sharePrice.

```
public class Company {

    public Company(String companyname,double currentSharePrice) {
        this.companyname=companyname;
        this.currentSharePrice=currentSharePrice;
    }

    public String companyname;
    public double currentSharePrice;
    public ArrayList<User> userList=new ArrayList<User>();

    public void registerUser(User U)
    {
        userList.add(U);
    }
    public void UnregisterUser(User U)
    {
        userList.remove(U);
    }
    public void ChangeShare(double Shareprice)
    {
        if(currentSharePrice!=Shareprice)
        {
            notifyUser();
        }
    }
    public void notifyUser()
    {
        for (User user : userList) {
            user.receiveNotice(this);
        }
    }
```

Create User Class(Observer):

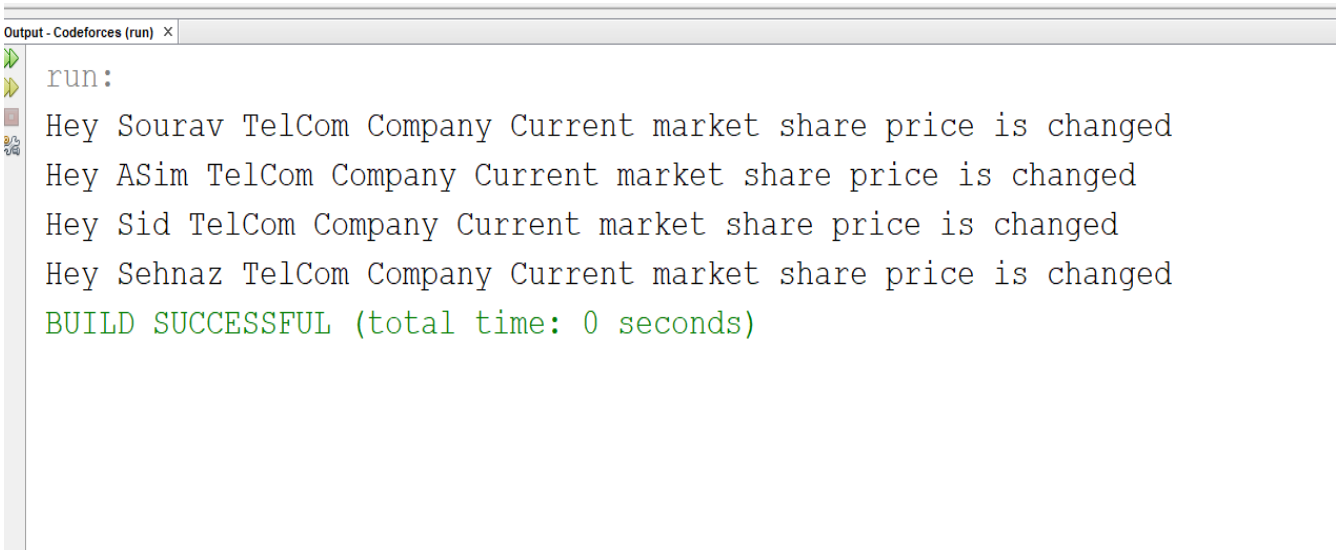
Next we will create Observer class, there will be a method called receiveNotify which is used for receive notification from Company and it will be alert user about that notification.

```
*/
public class User {
    public String name;

    public User(String name) {
        this.name = name;
    }

    public void receiveNotice(Company c)
    {
        System.out.println("Hey "+name+" "+c.companyname+" Company Current market share price is changed");
    }

}
```

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

```
public class Client {
    public static void main(String[] args) {
        Company c=new Company("TelCom",5000);
        User u1=new User("Sourav");
        User u2=new User("ASim");
        User u3=new User("Sid");
        User u4=new User("Sehnaz");
        c.registerUser(u1);
        c.registerUser(u2);
        c.registerUser(u3);
        c.registerUser(u4);
        c.ChangeShare(6000);


    }
}
```

When we run above program, we get following output.



1 SOLID Principle Observer Design Pattern Covers:

I think it follows OCP because we can extend the the code with new observers in the future rather than modifying existing code to make these new observers fit in. It also follows ISP because the Subject and Observer interface is precise and small for the specific job that the observer/subject is meant to do.
In general, the most relevant SOLID principle related to Observer is the Open/Close Principle: Once we have written the code of the observed object, we need not change the code when we want additional observers to know it, yet it is easy to add such observers - this is exactly what "closed for modifications, open for extensions" mean.