CS355 Software Engineering Process

Lab Week 05

Design Pattern – Observer Pattern

In this lab, we are going to look at another software design pattern: 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.

There are four participants in the Observer pattern:

• Subject, which is used to register observers. Objects use this interface to register as observers and also to remove themselves from being observers.

• 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.

• 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.

• ConcreateObserver, maintains a reference to a ConcreteSubject object and implements the Observer interface. Each observer registers with a concrete subject to receive updates

Implementation

Next, we are going to implement the observer pattern via using a Sports Lobby example.

Sports Lobby. 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.

```
public interface Subject {
        public void subscribeObserver(Observer observer);
        public void unSubscribeObserver(Observer observer);
        public void notifyObservers();
        public String subjectDetails();

}
```

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.

```
public interface Observer {
        public void update(String desc);
        public void subscribe();
        public void unSubscribe();

}
```

• 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.

```
public interface Commentary {
	public void setDesc(String desc);

}
```

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.

```
import java.util.List;

public class CommentaryObject implements Subject,Commentary{
	private final List<Observer>observers;
	private String desc;
	private final String subjectDetails;

	public CommentaryObject(List<Observer>observers,String subjectDetails){
		this.observers = observers;
		this.subjectDetails = subjectDetails;
	}

	@Override
	public void subscribeObserver(Observer observer) {
		observers.add(observer);

	}

//see observer.zip for full source code.
```

Note that CommentaryObject implements both interfaces Subject and Commentary. Therefore, it overrides all abstract methods defined in both interfaces. It also stores the reference to the observers registered to it.

Now we can construct a 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. [See SMSUsers.java for the full source code].

Exercise 1: Download the java files (Commentary.java, CommentaryObject.java, Observer.java, SMSUsers.java, Subject.java, TestObserver.java) for today's lab from Moodle, compile and run the code. Try to understand the relationships among difference classes.

Exercise 2: Add a new feature to Sports Lobby that allows commercial to be played. In other words, you need to create another interface commercial. In this interface, define a method called: setCommertical(String title). Then, you need to implement this interface in the CommentaryObject.

Exercise 3: Test your Commerical service in TestObserver.java. You may use the code in the TestOberver.java as a template.