Observer Pattern

Introduction

1. A newspaper publisher goes into business and begins publishing newspapers.
2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
3. You unsubscribe when you don't want papers anymore, and they stop being delivered.
4. While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

Definition: **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

[The power of **Loose Coupling**]

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Design Principle: Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Example – by ourselves

Weather Station

Observer Interface

```
/**
 * The Observer interface is implemented by all observers,
 * so they all have to implement the update() method.
 */
public interface Observer {
     public void update(float temp, float humidity, float pressure);
}
```

Subject Interface

```
/**
 * A subject should support three operations
 * register an observer
 * remove an observer
 * notify observers
 */
public interface Subject {
	public void registerObserver(Observer o);
	public void removeObserver(Observer o);
	public void notifyObservers();
}
```

DisplayElement Interface

```
public interface DisplayElement {
	public void display();
}
```

Current Condition Display class

An observer – can display weather information in its own format.

```
public class CurrentConditionDisplay implements Observer, DisplayElement{
	private float currentTemperature;
	private float currentHumidity;
	private float currentPressure;
	private Subject weatherData;

	public CurrentConditionDisplay(Subject weatherData) {
		this.weatherData = weatherData;
		weatherData.registerObserver(this);
	}
	@Override
	public void display() {
		System.out.println("Current Temperature:" + currentTemperature);
		System.out.println("Current Humidity:" + currentHumidity);
		System.out.println("Current Pressure:" + currentPressure);
	}
	@Override
	public void update(float temp, float humidity, float pressure) {
		this.currentTemperature = temp;
		this.currentHumidity = humidity;
		this.currentPressure = pressure;
		display();
	}
}
```

Weather Data class

A subject – can register observers, remove observers, modify weather data and notify all the registered observers.

Every time we set a new weather data, all the observers refresh their displays.

```
public class WeatherData implements Subject{
	private float temperature;
	private float humidity;
	private float pressure;
	private List<Observer> observers;

	public static void main(String[] args) {
		WeatherData weatherData = new WeatherData();
		CurrentConditionDisplay ccd = new CurrentConditionDisplay(weatherData);
		weatherData.setMeasurements(54, 23, 6);
		weatherData.setMeasurements(67, 2, 66);
		weatherData.setMeasurements(12, 88, 69);
	}
	public WeatherData() {
		observers = new ArrayList<Observer>();
	}
	@Override
	public void registerObserver(Observer o) {
		observers.add(o);
	}
	@Override
	public void removeObserver(Observer o) {
		if (observers.contains(o)) {
			observers.remove(o);
		}
	}
	@Override
	public void notifyObservers() {
		for (Observer o : observers) {
			o.update(temperature, humidity, pressure);
		}
	}
	public void measurementsChanged() {
		notifyObservers();
	}
	public void setMeasurements(float temperature, float humidity, float pressure)
	{
		this.temperature = temperature;
		this.humidity = humidity;
		this.pressure = pressure;
		measurementsChanged();
	}
}
```

Example – using Java's built-in Observer Pattern

Java built-in Observer Pattern

Java.util.Observable – used for creating Subject

Java.util.Observer – used for creating Observer

Weather Station

Observer

```
import java.util.Observable;
import java.util.Observer;

import com.keran.designpattern.observerPattern.DisplayElement;

public class CurrentConditionDisplay implements Observer, DisplayElement {

	private float currentTemperature;
	private float currentHumidity;
	private float currentPressure;
	Observable observable;

	public CurrentConditionDisplay(Observable observable) {
		this.observable = observable;
		observable.addObserver(this);
	}

	@Override
	public void update(Observable arg0, Object arg1) {
		if (arg0 instanceof WeatherData) {
			this.currentTemperature = ((WeatherData) arg0).temperature;
			this.currentHumidity = ((WeatherData) arg0).humidity;
			this.currentPressure = ((WeatherData) arg0).pressure;
			display();
		}
	}

	@Override
	public void display() {
		System.out.println("Current Temperature:" + currentTemperature);
		System.out.println("Current Humidity:" + currentHumidity);
		System.out.println("Current Pressure:" + currentPressure);
	}

}
```

Subject

```
import java.util.Observable;

public class WeatherData extends Observable {

	public float temperature;
	public float humidity;
	public float pressure;

	public static void main(String[] args) {
		WeatherData weatherData = new WeatherData();
		CurrentConditionDisplay ccd = new CurrentConditionDisplay(weatherData);
		weatherData.setMeasurements(54, 23, 6);
		weatherData.setMeasurements(67, 2, 66);
		weatherData.setMeasurements(12, 88, 69);
	}

	public WeatherData() {
	}

	public void measurementsChanged() {
		setChanged();
		notifyObservers();
	}

	public void setMeasurements(float temperature, float humidity, float pressure)
	{
		this.temperature = temperature;
		this.humidity = humidity;
		this.pressure = pressure;
		measurementsChanged();
	}

}
```

Why setChanged() ?

The setChanged() method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers.

If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified.

The setChanged() method is meant to give us more flexibility in how we update observers by allowing us to optimize the notification. For example, in our Weather Station, imagine if our measurements were so sensitive that the temperature readings were constantly fluctuating by a few tenths of a degree. That might cause the WeatherData object to send out notifications constantly. Instead, we might want to send out notifications only if the temperature changes more than half a degree and we could call setChanged() only after that happened.