Gözlemci(Observer) Tasarım Deseni

Nedir?

Gözlemci(Observer) tasarım deseni, **behavioral** tasarım desenlerinden biridir. Nesneler arasında **one-to-many** ilişki sağlar. Bir nesne durumunu **değiştirdiğinde,** ona bağlı **diğer** tüm nesneler uyarılır ve otomatik olarak **güncellenir.**

Ne zaman Kullanılır?

Bir nesnenin durumunun değişmesi ile o nesneye bağlı diğer nesnelerin bu değişimi bilmesini istiyorsak, böyle durumlarda bu tasarım desenini kullanabiliriz.

Nasıl Kullanılır?

Öncelikle iki tane interface sınıfına ihtiyacımız vardır: **Subject** ve **Observer. Subject** interface sınıfı durumu değişecek nesneyi temsil ederken, **Observer** türünden olan nesneler ise **Subject** türündeki nesneyi gözlemleyecek ve bir değişiklik olduğu zaman uyarılacaktır. Yani burada **Subject** etkileyen nesneyi, **Observer** ise etkilenen nesneleri temsil eder.

Faydaları Nedir?

1. **Loosely-coupled** uygulamalar yapmayı sağlar. **Subject** ile **Observer** birbirleriyle **loosely-coupled**'tır.
2. Bir nesnenin birden çok nesneyi otomatik olarak etkilemesini istiyorsak bu tasarım desenini kullanabiliriz. Örneğin, uygulamamızda A ve B kısımları olsun. A kısmında anlık sıcaklığın gösterildiğini varsayalım. B kısmı ise sıcaklık 20 derecenin altında olduğu zaman yeşil bir ışık göstersin. B kısmının sıcaklık değişikliklerine tepki göstermesi için kendisini A kısmının dinleyicisi(listener) olarak kaydetmesi gerekir. Kaydettikten sonra her bir sıcaklık değişimini izleyerek yeşil ışık gösterip göstermeyeceğini otomatik olarak kontrol eder.

Gerekenler

Türü interface olan **Subject** ve **Observer** sınıfları
Tek bir tane **somut Subject** sınıfı.
En az iki tane **somut Observer** sınıfı.
Test sınıfı

Not: Observer tasarım deseni **one-to-many** prensibini uyguladığı için tek bir tane **Subject** olmalı, birden çok ise **Observer** sınıfı olmalıdır.

Örnek Kullanım Alanları

1. **Model-View-Controller (MVC)** frameworklerinde kullanılır. **Observer** Views'leri temsil ederken, Model ise **Subject**'i temsil eder.
2. **java.util.EventListener** sınıfının tüm implementasyonlarında kullanılır.
3. javax.servlet.http.HttpSessionBindingListener
4. javax.servlet.http.HttpSessionAttributeListener
5. javax.faces.event.PhaseListener

Örnek Uygulama

Observer Interface

```
1   public interface Observer {
2        public void update(float temp, float humidity, float pressure);
3   }
```

Observer interface içerisinde sadece **tek** bir tane metod bulunması **yeterlidir.** Bizim örneğimizde **update()** isimli metod kullanıldı.

Subject Interface

```
1   public interface Subject {
```

```
2       public void registerObserver(Observer o);
3       public void removeObserver(Observer o);
4       public void notifyObservers();
5   }
```

Subject interface içerisinde bulunan tüm metodlar **zorunludur.** Bunların dışında başka metodlar eklenebilir.

DisplayElement Interface

```
1   public interface DisplayElement {
2       public void display();
3   }
```

Bu interface sınıfını kullanmak **zorunlu değildir.** Örneğimizde gerektiği için kullanıldı.

Somut Subject Sınıfı

```
1   public class WeatherData implements Subject {
2       private ArrayList observers;
3       private float temperature;
4       private float humidity;
5       private float pressure;
6
7       public WeatherData() {
8           observers = new ArrayList();
9       }
10
11      public void registerObserver(Observer o) {
12          observers.add(o);
13      }
14
15      public void removeObserver(Observer o) {
16          int i = observers.indexOf(o);
17          if (i >= 0) {
18              observers.remove(i);
19          }
20      }
21
22      public void notifyObservers() {
23          for (int i = 0; i < observers.size(); i++) {
24              Observer observer = (Observer)observers.get(i);
25              observer.update(temperature, humidity, pressure);
26          }
27      }
28
29      public void measurementsChanged() {
30          notifyObservers();
31      }
32
33      public void setMeasurements(float temperature, float humidity, float pressure) {
34          this.temperature= temperature;
35          this.humidity= humidity;
36          this.pressure= pressure;
37          measurementsChanged();
38      }
39
40      public float getTemperature() {
41          return temperature;
42      }
43
44      public float getHumidity() {
45          return humidity;
46      }
47
48      public float getPressure() {
49          return pressure;
```

```
50        }
51    }
```

WeatherData sınıfı **Observer** tasarım deseninde **somut Subject** sınıfını temsil eder.

Somut Observer Sınıfları

```
1     public class ForecastDisplay implements Observer, DisplayElement {
2          private float currentPressure = 29.92f;
3          private float lastPressure;
4          private WeatherData weatherData;
5
6          public ForecastDisplay(WeatherData weatherData) {
7              this.weatherData= weatherData;
8              weatherData.registerObserver(this);
9          }
10
11         public void update(float temp, float humidity, float pressure) {
12                     lastPressure = currentPressure;
13             currentPressure = pressure;
14
15             display();
16         }
17
18         public void display() {
19             System.out.print("Forecast: ");
20             if (currentPressure > lastPressure) {
21                 System.out.println("Improving weather on the way!");
22             } else if (currentPressure == lastPressure) {
23                 System.out.println("More of the same");
24             } else if (currentPressure < lastPressure) {
25                 System.out.println("Watch out for cooler, rainy weather");
26             }
27         }
28    }
```

Somut **Observer** sınıfı. Görüldüğü gibi constructor parametre olarak **somut Subject** türünden bir nesne alıyor ve bu nesnenin **registerObserver()** metodunu kullanarak kendisini kaydediyor. **update()** metodunda ise **Subject** sınıfının yaptığı değişikliği alıyor ve işliyor.

```
1     public class StatisticsDisplay implements Observer, DisplayElement {
2          private float maxTemp = 0.0f;
3          private float minTemp = 200;
4          private float tempSum= 0.0f;
5          private int numReadings;
6          private WeatherData weatherData;
7
8          public StatisticsDisplay(WeatherData weatherData) {
9              this.weatherData= weatherData;
10             weatherData.registerObserver(this);
11         }
12
13         public void update(float temp, float humidity, float pressure) {
14             tempSum += temp;
15             numReadings++;
16
17             if (temp > maxTemp) {
18                 maxTemp = temp;
19             }
20
21             if (temp < minTemp) {
22                 minTemp = temp;
23             }
24
25             display();
```

```
26        }
27
28        public void display() {
29            System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings)
30                + "/" + maxTemp + "/" + minTemp);
31        }
32    }
```

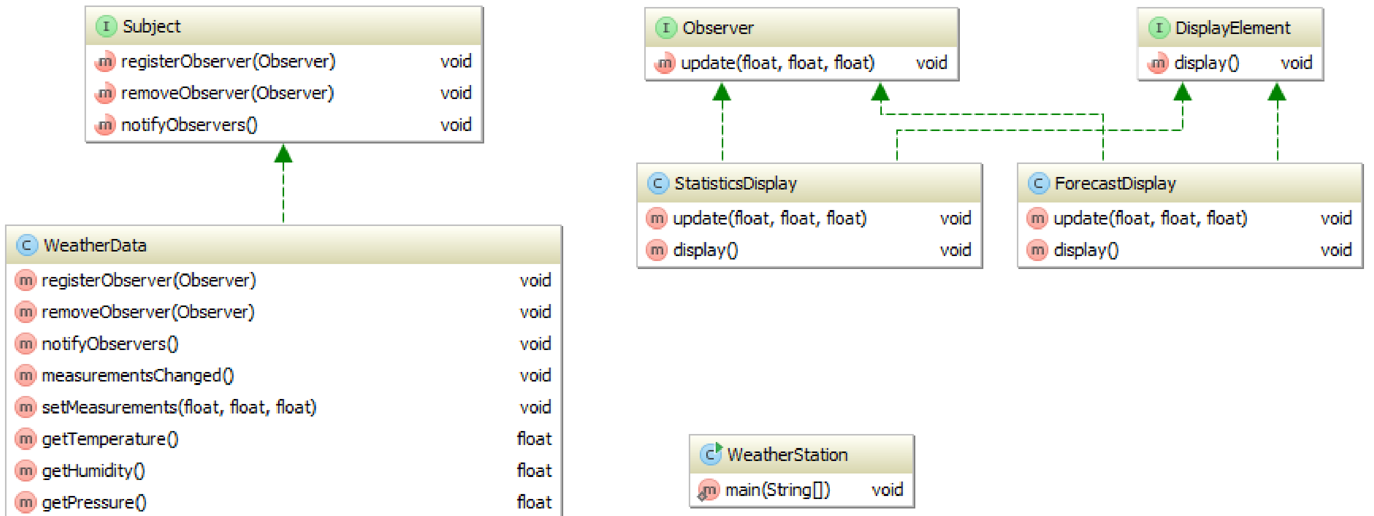
İkinci Somut **Observer** sınıfı.

Test Sınıfı

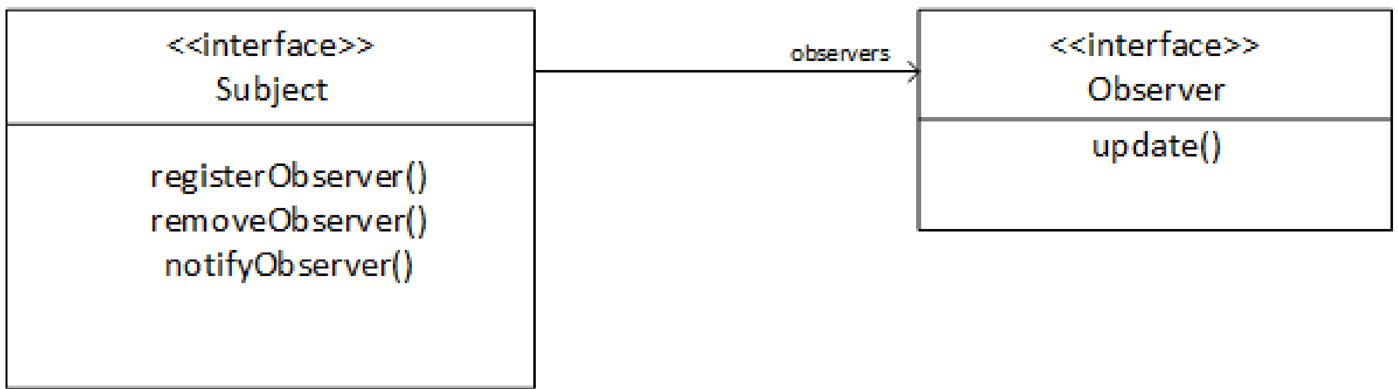
```
1   public class WeatherStation {
2
3       public static void main(String[] args) {
4           WeatherData weatherData = new WeatherData();
5
6           StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
7           ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);
8
9           weatherData.setMeasurements(80, 65, 30.4f);
10          weatherData.setMeasurements(82, 70, 29.2f);
11          weatherData.setMeasurements(78, 90, 29.2f);
12      }
13  }
```

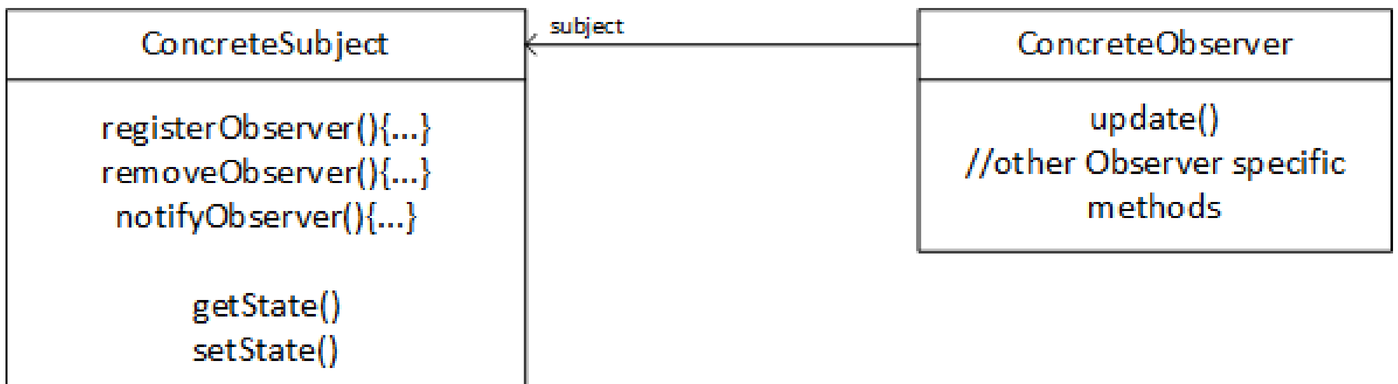
Test sınıfı. Dikkat edersek, observer sınıflarından nesne yaratılırken, parametre olarak **somut Subject** sınıfı kullanıldı.

Örneğimizin UML Diagramı



Soyut Fabrika Tasarım Deseni'nin Şematik Gösterimi


<<interface>>
Subject
registerObserver()
removeObserver()
notifyObserver()
observers
<<interface>>
Observer
update()


ConcreteSubject
registerObserver(){...}
removeObserver(){...}
notifyObserver(){...}
getState()
setState()
subject
ConcreteObserver
update()
//other Observer specific
methods