设计模式: 观察者（Observer）模式

实验编号 exp03

0.1 实验目的

1. 理解设计模式之观察者（Observer）模式。[1]
2. 理解模式中所包含的 Java 设计原则。
3. 基于观察者（Observer）模式编写 Java 程序，加深对观察者模式的理解。
4. 学习使用 Java 自带的观察者模式实现。

0.2 需求分析

假设我们团队获得了一项合约，是需求方 Weather-O-Rama 公司委托我们开发一个气象站应用。首先来看一下需求方 Weather-O-Rama 公司对气象站应用的需求概况（如图 1）。其中，WeatherData 对象用于从气象站获取温度、湿度和气压数据，并驱动多个显示装置显示。合约中要求至少包括目前状况、气象统计、天气预报的显示布告板。同时，需求方所给的 WeatherData 类参考设计如下。

Class: WeatherData

```
public class WeatherData {
  public float getTemperature() {
    // 从气象站获取温度数据，由需求方提供方法
  }
  public float getHumidity() {
    // 从气象站获取湿度数据，由需求方提供方法
  }
  public float getPressure() {
    // 从气象站获取气压数据，由需求方提供方法
  }
  public void measurementsChanged() {
```

[1]接下来两章分别讲解并示范两个 Java 经典的设计模式。设计这两个章节的目的是：对 Java 中接口、抽象类、继承、多态等概念的更进一步理解，它们有什么用？搞得这么复杂干什么？有些东西不写代码是学不来的，也体会不到。

```
    // 当气象数据值发生改变时，该方法被期望调用
  }
}
```

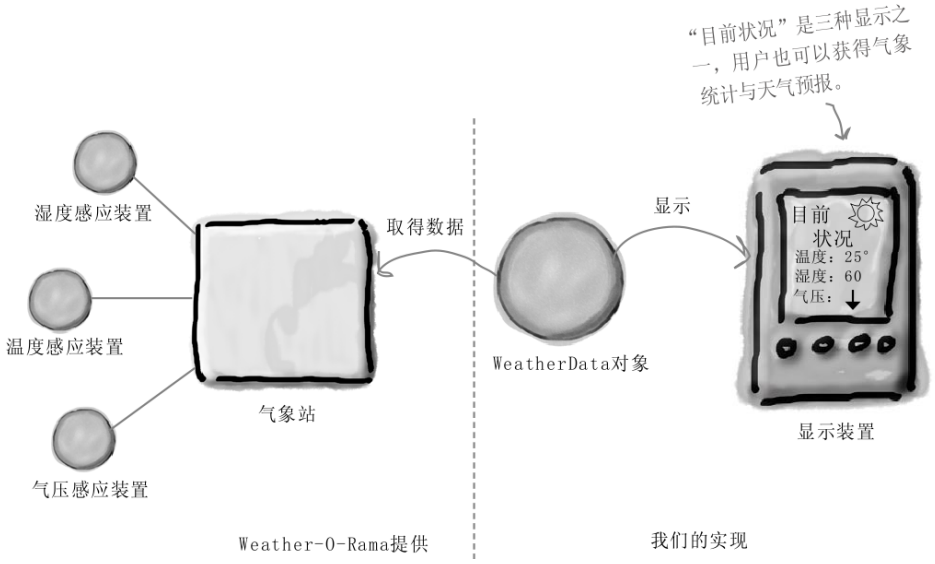

图 1: 气象站应用的架构

根据任务合约中的内容和需求方提供的 WeatherData 类参考，你一般会这样设计这个气象观测站：在 measurementsChanged 方法中添加了功能代码，实现了对三个布告板的数据更新，如下所示。

```
public class WeatherData {
  // 实例变量声明

  public void measurementsChanged() {
    float temp = getTemperature(); // 调用取得三种数据的方法
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
  }
  // 这里是其他的 WeatherData 方法
}
```

上述实现方案非常不佳，我们来分析一下并从如下选项中找出上述实现所存在的问题：

- 针对具体实现编程，而非针对接口编程 ❒
- 对于每添加一个布告板，都得修改代码 ❒
- 无法在运行时动态增加（删除）布告板 ❒

- 布告板没有实现一个共同的接口 ❒
- 尚未封装改变的部分 ❒
- 侵犯了 WeatherData 类的封装 ❒

所以，本实验将采取观察者（Observer）模式来实现气象站应用。根据 UML 类图 2，实现气象站，要求能够避免上述分析中旧设计所存在的问题。

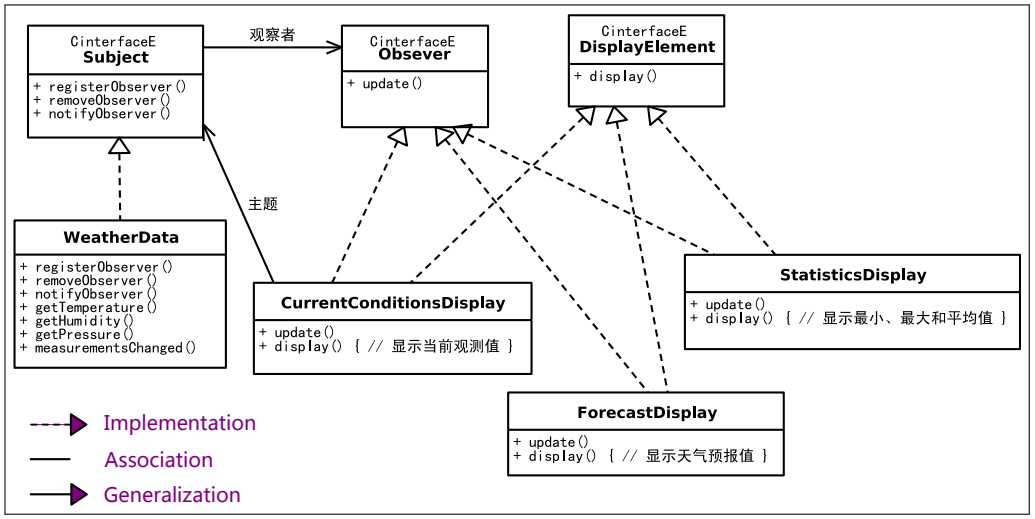

图 2: 气象站 UML 类图

0.3 问题分析

0.3.1 观察者模式定义

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新，如图 3。

在现实世界中，我们可以用报纸订阅服务来类比观察者模式。

0.3.2 松耦合

设计原则

为了交互对象之间的松耦合设计而努力。

两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

- 关于观察者的一切，主题只知道观察者实现了某个接口（也就是 Observer 接口）。主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。

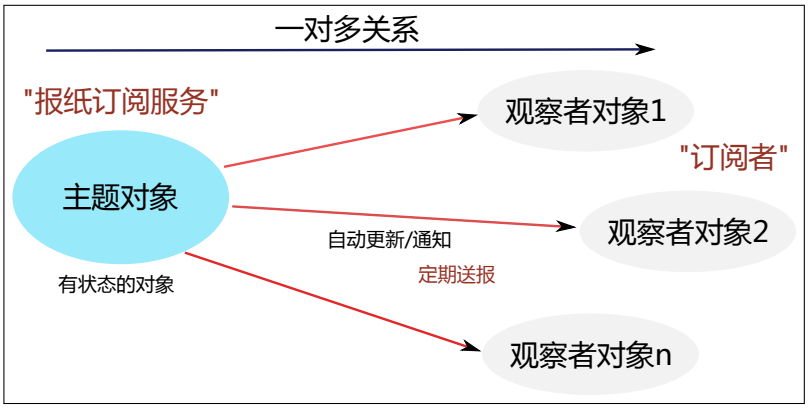

图 3: 观察者模式

- 任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现 Observer 接口的对象列表，所以我们可以随时增加观察者。事实上，在运行时我们可以用新的观察者取代现有的观察者，主题不会受到任何影响。同样的，也可以在任何时候删除某些观察者。
- 有新类型的观察者出现时，主题的代码不需要修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所有要做的就是在新的类里实现此观察者接口，然后注册为观察者即可。主题不在乎别的，它只会发送通知给所有实现了观察者接口的对象。
- 我们可以独立地复用主题或观察者。如果我们在其他地方需要使用主题或观察者，可以轻易地复用，因为二者并非紧耦合。
- 改变主题或观察者其中一方，并不会影响另一方。因为两者是松耦合的，所以只要他们之间的接口仍被遵守，我们就可以自由地改变他们。

0.4 实验过程、步骤及原始记录

实验过程和代码如下：

```
1 .
2 .
3 .
4 .
5 .
6 .
```

输出结果参考如下（仅实现了当前气象状况布告板，另外的布告板请自行添加并完成实验报告）：

output

```
Current conditions: 80.0F degrees and 65.0% humidity
Current conditions: 78.0F degrees and 90.0% humidity
```

0.5 参考程序

本实验为学习验证性实验，直接给出参考程序，希望大家认真学习理解观察者模式的用法以及其中包含的设计哲学，并尝试增加其他类型的气象布告板，数据人工模拟即可。

File: Subject.java

```
package ouc.cs.java.desingpattern.observer;

public interface Subject {
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}
```

File: Observer.java

```
package ouc.cs.java.desingpattern.observer;

public interface Observer {
  public void update(float temperature, float humidity, float pressure);
}
```

File: DisplayElement.java

```
package ouc.cs.java.desingpattern.observer;

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

File: WeatherData.java

```
package ouc.cs.java.desingpattern.observer;

import java.util.ArrayList;

public class WeatherData implements Subject {

  private ArrayList observers;

  private float temperature;
  private float humidity;
  private float pressure;

  public WeatherData() {
    observers = new ArrayList();
  }

  @Override
  public void registerObserver(Observer o) {
    observers.add(o);
```

```
20    }
22    @Override
23    public void removeObserver(Observer o) {
24      int i = observers.indexOf(o);
25      if (i >= 0) {
26        observers.remove(i);
27      }
28    }
30    @Override
31    public void notifyObservers() {
32      for (int i = 0; i < observers.size(); i++) {
33        Observer observer = (Observer) observers.get(i);
34        observer.update(temperature, humidity, pressure);
35      }
36    }
38    public void measurementsChanged() {
39      notifyObservers();
40    }
42    public void setMeasurements(float temperature, float humidity, float pressure) {
43      this.temperature = temperature;
44      this.humidity = humidity;
45      this.pressure = pressure;
46      measurementsChanged();
47    }
48  }
```

File: CurrentConditionsDisplay.java

```
1   package ouc.cs.java.desingpattern.observer;
3   public class CurrentConditionsDisplay implements Observer, DisplayElement {
5     private float temperature;
6     private float humidity;
7     private Subject weatherData;
9     public CurrentConditionsDisplay(Subject weatherData) {
10      this.weatherData = weatherData;
11      weatherData.registerObserver(this);
12    }
14    @Override
15    public void update(float temperature, float humidity, float pressure) {
16      this.temperature = temperature;
17      this.humidity = humidity;
18      display();
19    }
```

```
21    @Override
22    public void display() {
23      System.out.println("Current conditions: " + temperature
24      + "F degrees and " + humidity + "% humidity");
25    }
26  }
```

File: WeatherStation.java

```
1  package ouc.cs.java.desingpattern.observer;

3  public class WeatherStation {

5    public static void main(String[] args) {
6      WeatherData weatherData = new WeatherData();
7      CurrentConditionsDisplay cd = new CurrentConditionsDisplay(weatherData);

9      weatherData.setMeasurements(80, 65, 30.4f);
10     weatherData.setMeasurements(78, 90, 29.2f);
11   }
12 }
```

0.6 Java 自带的观察者模式

由于观察者模式的广泛应用性，Java API 自带了观察者模式的实现。其中，Observable 对象是被观察者，Observer 对象是观察者。使用 Java 自带的 API 实现观察者模式非常简单：

- 创建被观察者类（即 Subject），它继承自 java.util.Observable 类；
- 创建观察者类，它实现 java.util.Observer 接口；

0.6.1 被观察者类

使用被观察者类的 addObserver() 方法把观察者对象添加到观察者对象列表中。当被观察者中的事件发生变化时执行 setChanged() 和 notifyObservers() 方法。

- setChange() 方法用来设置一个内部标志位注明数据发生了变化；
- notifyObservers() 方法会去调用观察者对象列表中所有的 Observer 的 update() 方法，通知它们数据发生了变化。

注意：只有在 setChange() 被调用后，notifyObservers() 才会去调用 update()。

0.6.2 观察者类

观察者类需要实现 Observer 接口的唯一方法 update()。

```
void update(Observable o, Object arg);
```

形参 Object arg 对应一个由 notifyObservers(Object arg) 传递来的参数，当执行的是 notifyObservers() 时，arg 为 null。

0.6.3 示例代码

被观察者 ServerManager

```
package ouc.cs.java.test.observer;

import java.util.Observable;

/**
* 被观察者
*
*/
public class ServerManager extends Observable {

  private int data = 0;

  public int getData() {
    return data;
  }

  public void setData(int i) {

    /*
    * 只有在setChanged()被调用后，notifyObservers()才会去调用update()，否则什么都不干
    */
    if (this.data != i) {
      this.data = i;
      setChanged();
    }
    notifyObservers();
  }
}
```

观察者 ObserverA

```
package ouc.cs.java.test.observer;

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

/**
* 观察者A
*
*/
public class ObserverA implements Observer {
```

```
12      public ObserverA(ServerManager sm) {
13        super();
14        sm.addObserver(this);
15      }

17      public void update(Observable arg0, Object arg1) {
18        System.out.println("ObserverA receive: Data has changed to " + ((ServerManager) arg0).getData());
19      }
20    }
```

观察者 ObserverB

```
1   package ouc.cs.java.test.observer;

3   import java.util.Observable;
4   import java.util.Observer;

6   /**
7   * 观察者B
8   *
9   */
10  public class ObserverB implements Observer {

12    public ObserverB(ServerManager sm) {
13      super();
14      sm.addObserver(this);
15    }

17    public void update(Observable arg0, Object arg1) {
18      System.out.println("ObserverB receive: Data has changed to " + ((ServerManager) arg0).getData());
19    }
20  }
```

测试主程序 TestJavaObserver

```
1   package ouc.cs.java.test.observer;

3   public class TestJavaObserver {

5     public static void main(String[] args) {

7       ServerManager sm = new ServerManager();
8       ObserverA a = new ObserverA(sm);
9       ObserverB b = new ObserverB(sm);
10      sm.setData(5);
11      sm.setData(8);

13      /*
14      * 注销观察者，以后被观察者有数据变化就不再通知这个已注销的观察者
15      */
```

```
16          sm.deleteObserver(a);
17          sm.setData(10);
18        }
19      }
```