


观察者模式

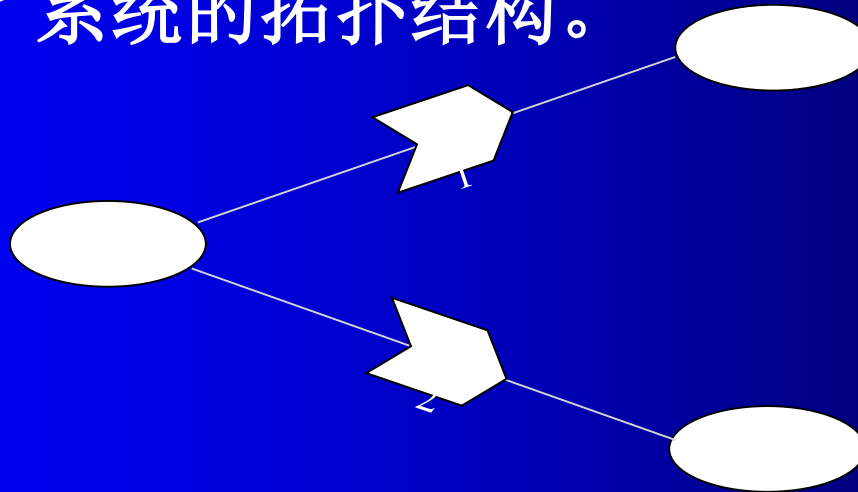


内容回顾

- 软件体系结构
- 软件设计模式

软件体系结构的定义(6)

- 一个软件系统的体系结构定义了组成系统的
 - 构件（components），
 - 连接件（connectors），和
 - 它们之间的匹配
- 这里，**构件**用于实施计算和保存状态，**连接件**用于表达构件之间的关系，**构件和连接件之间的匹配**表示了系统的拓扑结构。



什么是设计模式？

在我们写代码的过程中，如果一个问题反复发生，那么这个问题的解决方案就会被有效使用，这种被频繁使用的解决方案就叫做模式。

设计模式 (Design pattern) 是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码，让代码更容易被他人理解、保证代码可靠性。



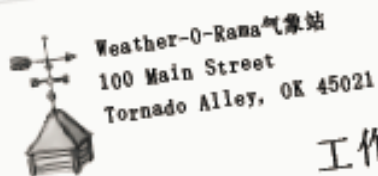
“

善用合理的设计模式开发程序模块，
能极大提高开发效率，
减少代码的耦合度，减少bug的产生，
能减少项目后期更新维护成本。

”

恭喜你！

你的团队刚刚赢得一纸合约，负责建立
Weather-O-Rama公司的下一代气象站——
Internet气象观测站。



工作合约

恭喜贵公司获选为敝公司建立下一代Internet气象观测站！
该气象站必须建立在我们专利申请中的WeatherData对象
上，由WeatherData对象负责追踪目前的天气状况（温度、
湿度、气压）。我们希望贵公司能建立一个应用，有三种
布告板，分别显示目前的状况、气象统计及简单的预报。
当WeatherObject对象获得最新的测量数据时，三种布告板
必须实时更新。

而且，这是一个可以扩展的气象站，Weather-O-Rama气象
站希望公布一组API，好让其他开发人员可以写出自己的
气象布告板，并插入此应用中。我们希望贵公司能提供这
样的API。

Weather-O-Rama气象站有很好的商业营运模式：一旦客
户上钩，他们使用每个布告板都要付钱。最好的部分就是，
为了感谢贵公司建立此系统，我们将以公司的认股权支付
你。

我们期待看到你的设计和应用的alpha版本。
真挚的

Johnny Hurricane

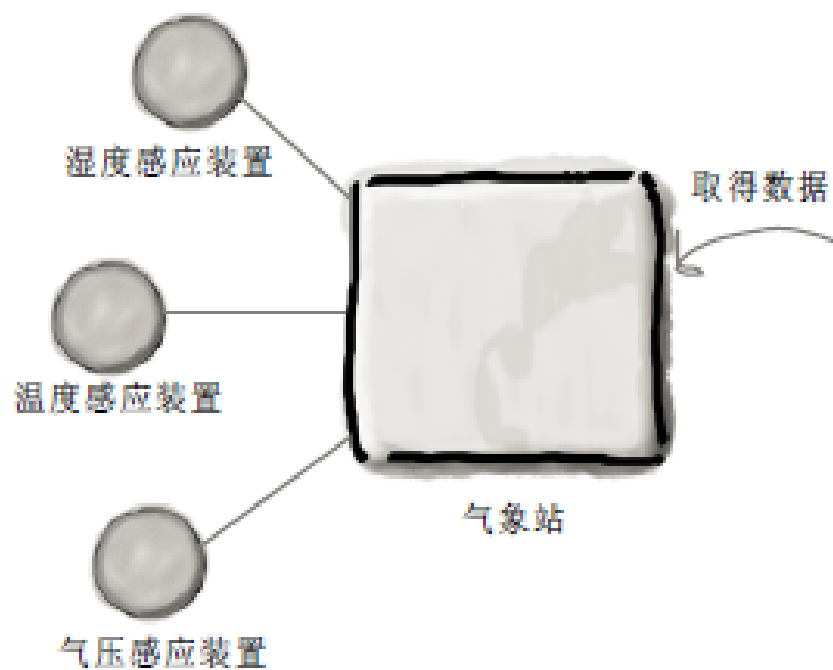
Johnny Hurricane——Weather-O-Rama气象站执行长

附注：我们正通宵整理WeatherData源文件给你们。

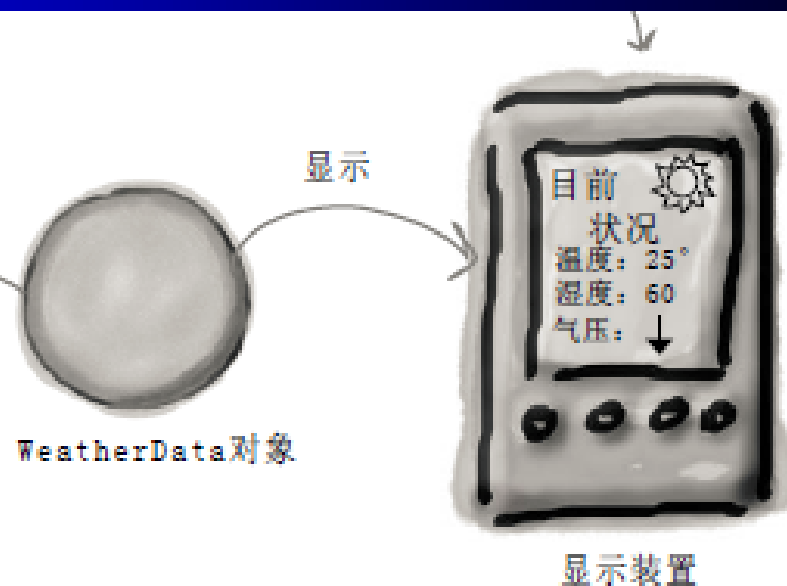
WeatherData

```
getTemperature()  
getHumidity()  
getPressure()  
measure-  
mentsChanged()
```





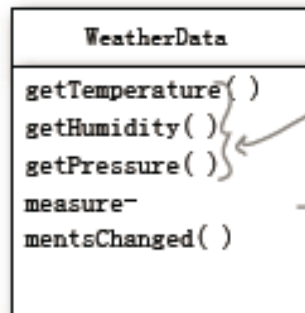
Weather-O-Rama提供



我们的实现

Weather对象知道如何跟物理气象站联系，以去打更新的数据。WeatherData对象会随即更新单个布告板的显示：目前状况（温度、湿度、气压）、气象统计和天气预报

如同他们所承诺的，隔天早上收到了WeatherData源文件，看了一下代码，一切都很直接：



这三个方法各自返回最近的气象测量数据
(分别为，温度、湿度、气压)。
我们不在乎这些变量“如何”被设置，
WeatherData对象自己知道如何从气象站获取更新
信息。

WeatherObject的开发人员留了一个
线索，好让我们知道该加些什么
.....

```
/*  
 * 一旦气象测量更新，此方法会被调用  
 */  
public void measurementsChanged() {  
    // 你的代码加在这里  
}
```

WeatherData.java

再次提醒，这只是三个显示
布告板中的一个。



显示装置

我们的工作是实现measurementsChanged()，好让它更新
目前状况、气象统计、天气预报的显示布告板。


```
public class WeatherData {
```

```
// 实例变量声明
```

```
public void measurementsChanged () {
```

```
float temp = getTemperature ();
```

```
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WeatherData方法
```

```
}
```

调用 WeatherData 的三个
getXxx() 方法，以取得最近的
测量值。这些getXxx() 方法已
经实现好了。

现在，更新布告
板……

调用每个布告板更新显示，
传入最新的测量。



Sharpen your pencil

在我们的第一个实现中，下列哪种说法正确？（多选）



A. 我们是针对具体实现编程，而非针对接口。



B. 对于每个新的布告板，我们都得修改代码。



C. 我们无法在运行时动态地增加（或删除）布告板。



D. 布告板没有实现一个共同的接口。



E. 我们尚未封装改变的部分。



F. 我们侵犯了WeatherData类的封装。

我们的实现有什么不合理？

```
public void measurementsChanged () {
```

```
    float temp = getTemperature ();  
    float humidity = getHumidity ();  
    float pressure = getPressure ();
```

```
    currentConditionsDisplay.update ( temp, humidity, pressure );  
    statisticsDisplay.update ( temp, humidity, pressure );  
    forecastDisplay.update ( temp, humidity, pressure );
```

```
}
```

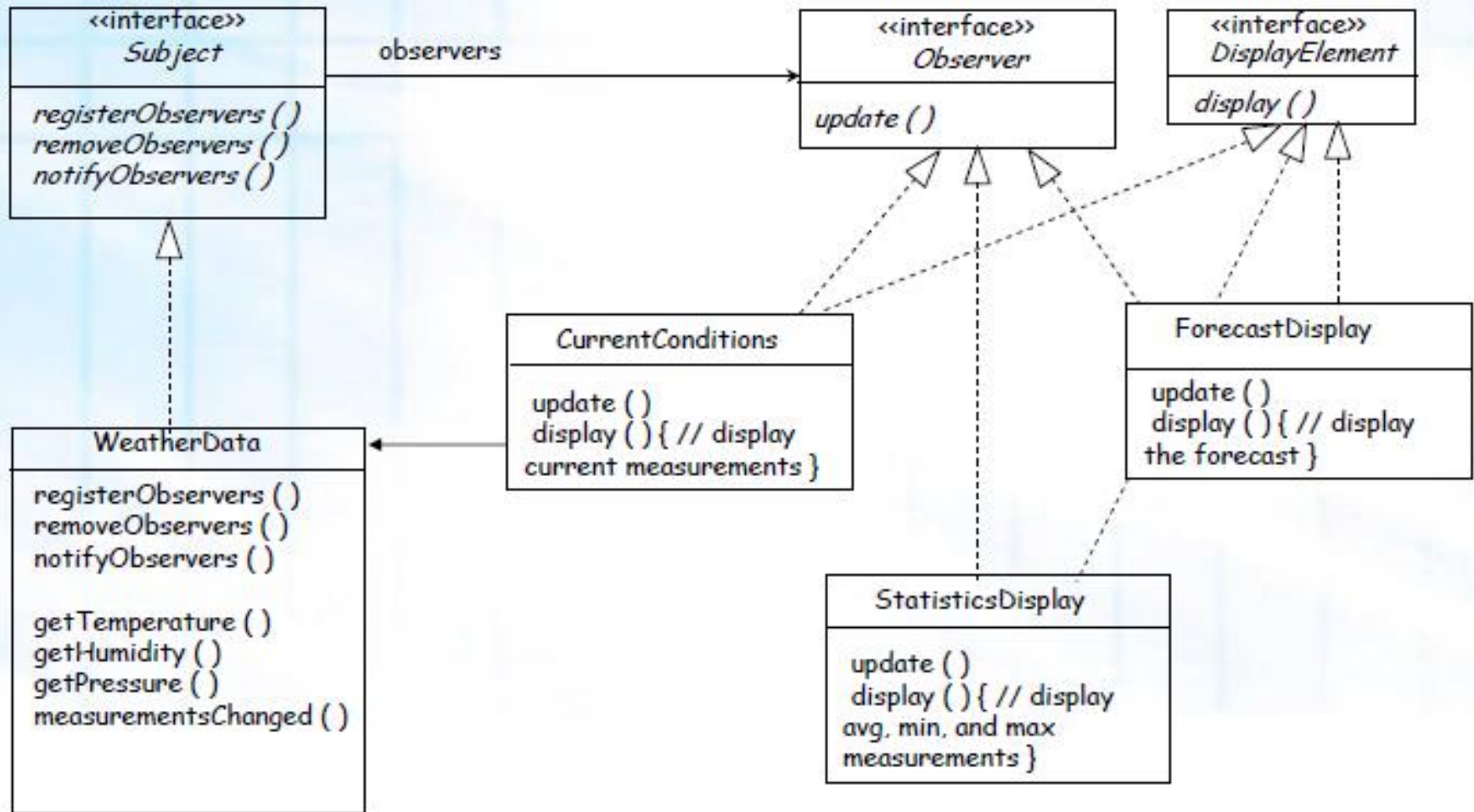
改变的地方，需要封装起来。

}

针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序。

至少，这里看起来像是一个统一的接口，布告板的方法名称都是update()，参数都是温度、湿度、气压。

Weatherstation设计



实现接口

Subject, Observer, DisplayElement

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

这两个方法可以让Observer注册或者删除。

当Subject的状态发声变化时调用这个方法
来通知所有的观察者（Observers）。

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

所有的观察者（Observers）都必须实现Observer接口，所以所有的它们必须实现update()方法。

这些值是在当测量值发生变化时观察者（Observers）从Subject获取的。

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

DisplayElement接口仅包括一个display()方法，当显示功能需要显示数据的时候我们可以调用这个方法。

在WeatherData中实现主题接口

```
import java.util.ArrayList;
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;
```

WeatherData现在实现了Subject接口。

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

增加了一个数组来获取观察者Observers，我们在构造函数中创建Observers。

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

```
    public void removeObserver (Observer o) {
        int j = observers.indexOf(o);
        if (j >= 0) {
            observers.remove(j);
        }
    }
```

```
    public void notifyObservers ( ) {
        for (int j = 0; j < observers.size(); j++) {
            Observer observer = (Observer)observers.get(j);
            observer.update(temperature, humidity, pressure);
        }
    }
```

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

当测量值发生变化时我们可以通知观察者observers。

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

```
    } //其他方法实现
```

这里我们实现了Subject中的空方法

实现当前环境显示

此布告板实现了Observer接口，所以
可以从WeatherData对象中获得改变。

它也实现了DisplayElement接口，
因为我们的API规定所有的布告
板都必须实现此接口。

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
```

```
    private float temperature;  
    private float humidity;  
    private Subject weatherData;
```

```
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }
```

构造器需要 weatherData对象（也
就是主题）作为注册之用。

```
    public void update(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        display();  
    }
```

当update()被调用时，我们
把温度和湿度保存起来，
然后调用display()。

```
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }
```

display()方法就只是
把最近的温度和湿
度显示出来。

启动气象站



❶ 先建立一个测试程序

气象站已经完成得差不多了，我们还需要一些代码将这一切连接起来。这是我们的第一次尝试，本书中稍后我们会再回来确定每个组件都能通过配置文件来达到容易“插拔”。现在开始测试吧：

```
public class WeatherStation {  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
  
        CurrentConditionsDisplay currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);  
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);  
  
        weatherData.setMeasurements(80, 65, 30.4f);  
        weatherData.setMeasurements(82, 70, 29.2f);  
        weatherData.setMeasurements(78, 90, 29.2f);  
    }  
}
```

如果你还不想下载完整的代码，可以将这两行注释掉，就能顺利执行了。

首先，建立一个WeatherData对象。

建立三个布告板，并把WeatherData对象传给它们。

模拟新的气象测量。

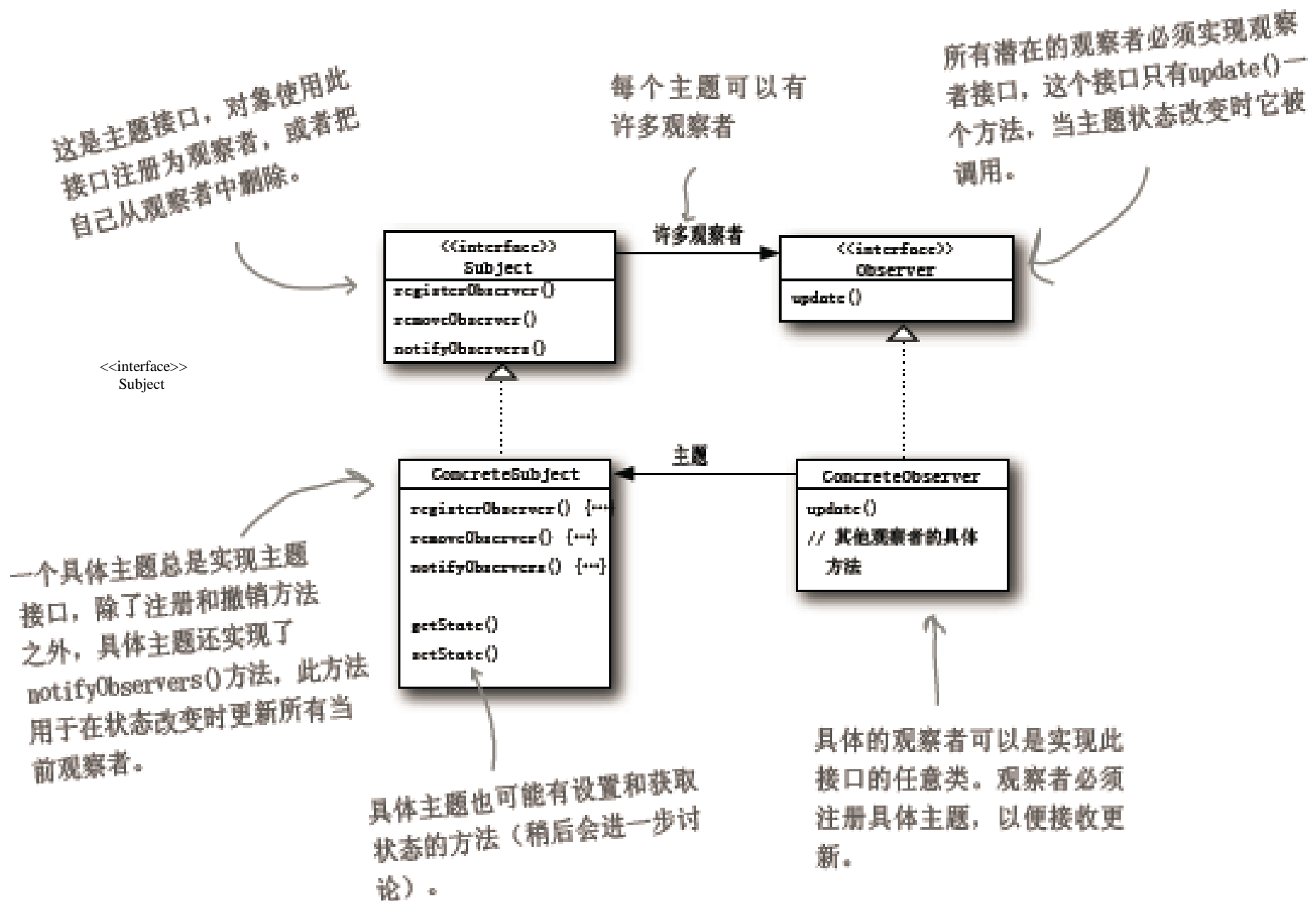
Johnny Hurricane (Weather-O-Rama气象站的CEO) 刚刚来电告知, 他们还需要酷热指数 (HeatIndex) 布告板, 这是不可或缺的。细节如下:

酷热指数是一个结合温度和湿度的指数, 用来显示人的温度感受。可以利用温度T和相对湿度RH套用下面的公式来计算酷热指数:

heatindex =

$$\begin{aligned} & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T \\ & * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} \\ & * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - \\ & 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH \\ & + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * \\ & 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3 \end{aligned}$$

定义观察者模式：类图



- **Subject**只需要知道有某个实现了确定的接口（指**Observers**接口）的观察者（**Observer**）就可以了。
 - 我们可以在任何时候增加观察者（**Observer**）。
 - 我们不需要修改**Subject**的代码来增加新类型的观察者（**Observer**）。
 - 我们可以分别重用**Subject**或者**Observer**。
 - 对于**Subject**和**Observer**的修改均不会影响对方。
- OK**，这里又涉及到了一个设计原则：



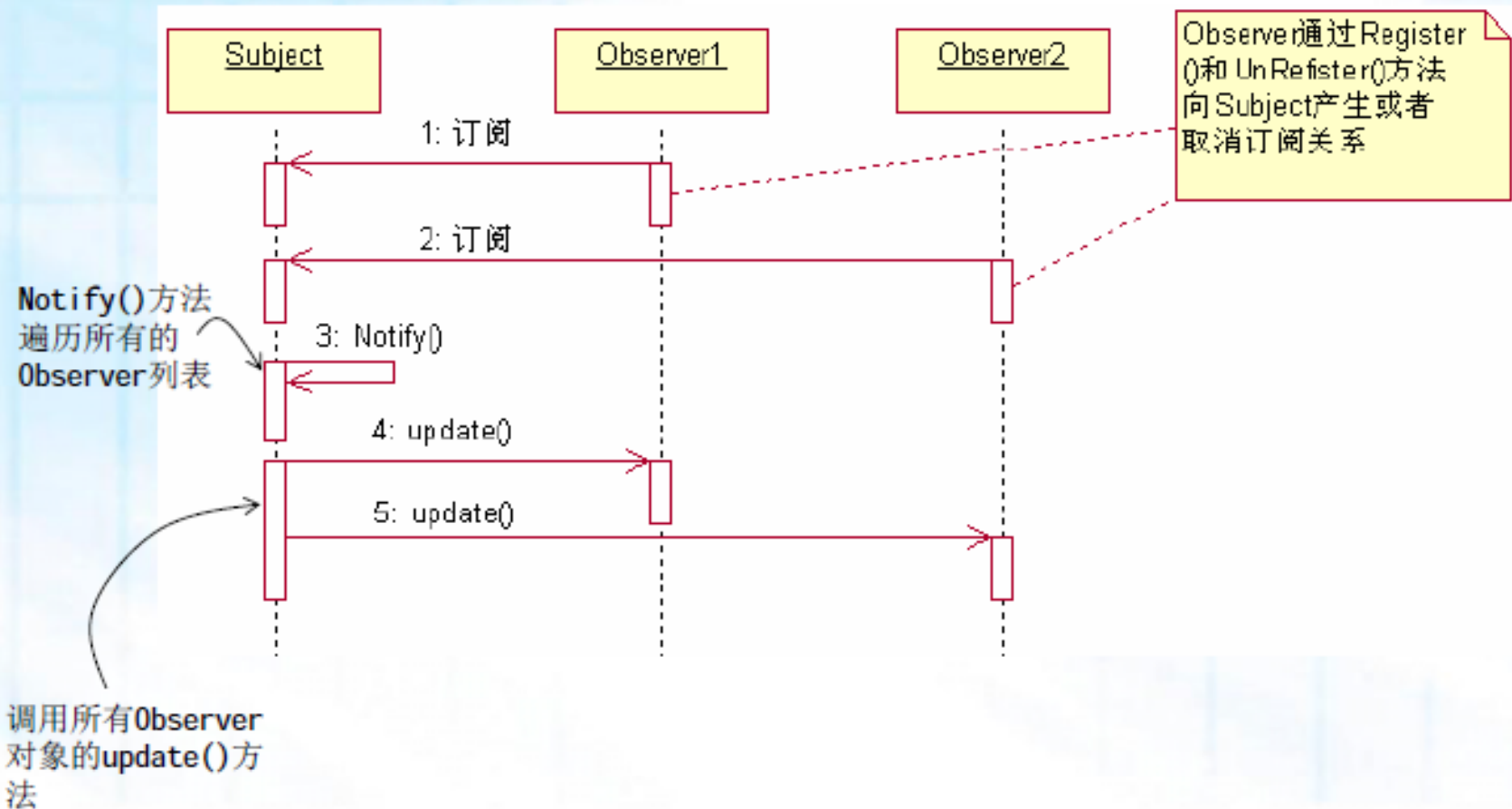
Design Principle

相互之间有影响的对象应尽量采用松耦合的设计

Strive for loosely coupled design between objects that interact.

松耦合的设计使得我们可以建立富有柔性的面向对象的系统，因为在对象之间的影响度很小，所以便于扩展和修改，

观察者模式涉及到的类的交互图



认识观察者模式

我们看看报纸和杂志的订阅是怎么回事：

- ❶ 报社的业务就是出版报纸。
- ❷ 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订户，你就会一直收到新报纸。
- ❸ 当你不想再看报纸的时候，取消订阅，他们就不会再送新报纸来。
- ❹ 只要报社还在运营，就会一直有人（或单位）向他们订阅报纸或取消订阅报纸。

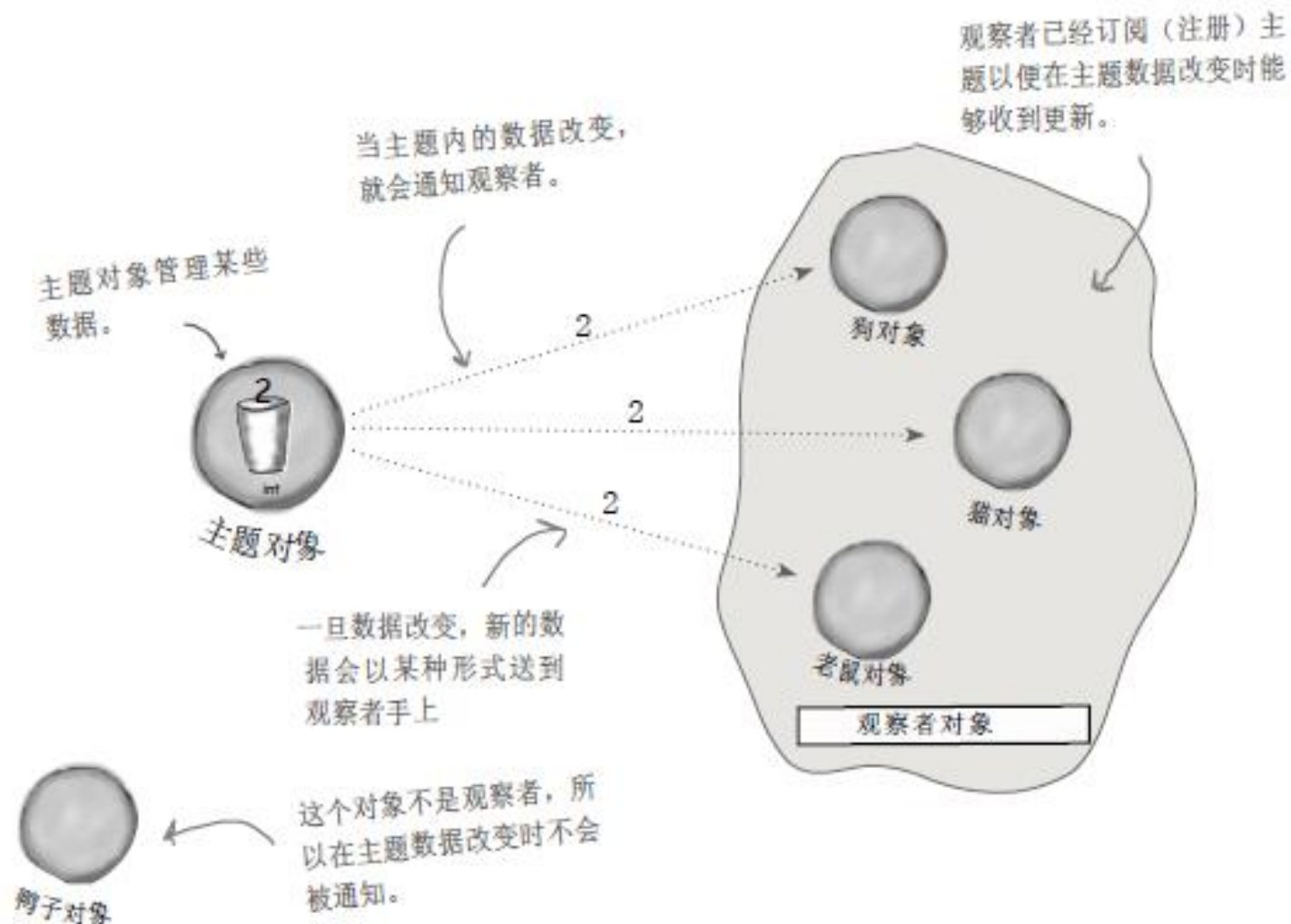
担心错过对象村的重大事件吗？才不会呢！
我们可是订了报的！



出版者+订阅者=观察者模式

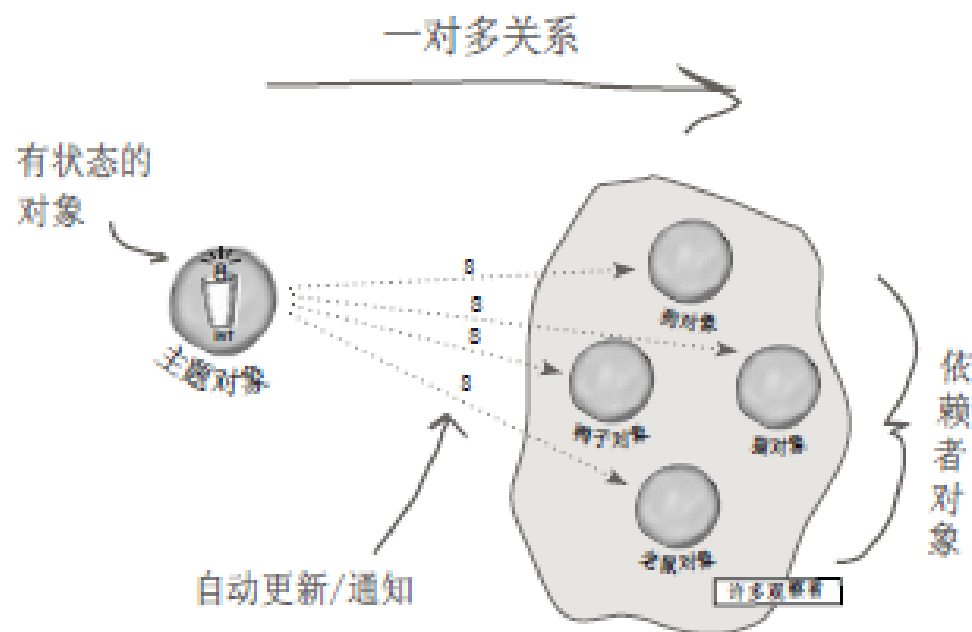
如果你了解报纸的订阅是怎么回事，其实就知道观察者模式是怎么回事，只是名称不太一样：出版者改称为“主题”（Subject），订阅者改称为“观察者”（Observer）。

让我们看得更仔细一点：



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。

让我们看看这个定义，并和之前的例子做个对照：



观察者模式定义了一系列对象之间的一对多关系。

当一个对象改变状态，其他依赖者都会收到通知。