软件建模与分析
Software Modeling and Analyzing

主讲人：徐峰磊

观察者（Observer）模式

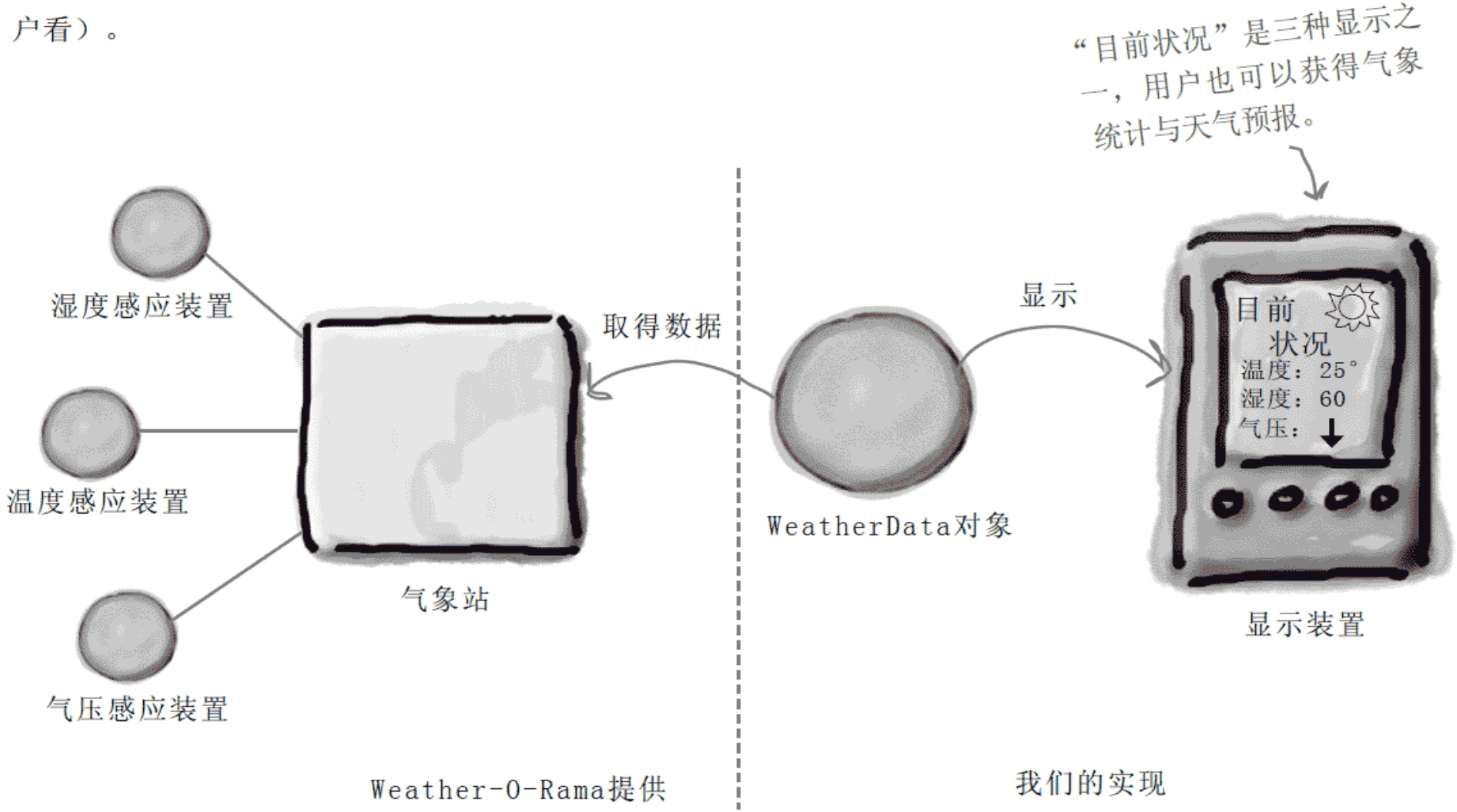
- 让你的对象知悉现况
- 有一个模式可以帮你的对象知悉现况，不会错过该对象感兴趣的事。对象甚至在运行时可决定是否要继续被通知。观察者模式是JDK中使用最多的模式之一，非常有用。我们也会一并介绍一对多关系，以及松耦合（对，没错,我们说耦合）。有了观察者，你将会消息灵通。

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**。

气象监测应用的概况

此系统中的三个部分是气象站（获取实际气象数据的物理装置）、WeatherData对象（追踪来自气象站的数据，并更新布告板）和布告板（显示目前天气状况给用户看）。

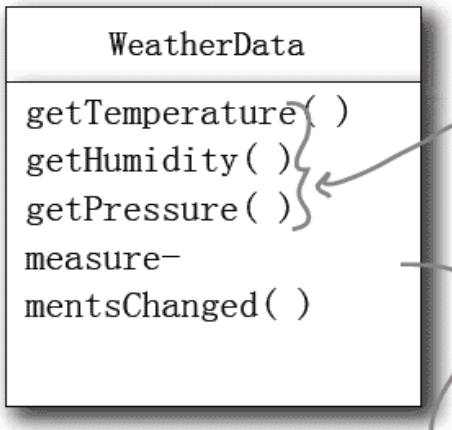


下一代Internet气象观测站

- **WeatherData**对象知道如何跟物理气象站联系，以取得更新的数据。**WeatherData**对象会随即更新三个布告板的显示：目前状况（温度、湿度、气压）、气象统计和天气预报。
- “目前状况”是三种显示之一， 用户也可以获得气象统计与天气预报。如果我们选择接受这个项目，我们的工作就是建立一个应用，利用**WeatherData**对象取得数据，并更新三个布告板：目前状况、气象统计和天气预报。

瞧一瞧刚送到的WeatherData类

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

mentsChanged( )

```
/*
 * 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

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

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。

Weather-O-Rama气象站的要求说明并不是很清楚，我们必须搞懂该做些什么。那么，我们目前知道些什么呢？

- WeatherData类具有getter方法，可以取得三个测量值：温度、湿度与气压。
- 当新的测量数据备妥时，measurementsChanged()方法就会被调用（我们不在乎此方法是如何被调用的，我们只在乎它被调用了）。
- 我们需要实现三个使用天气数据的布告板：“目前状况”布告、“气象统计”布告、“天气预报”布告。一旦WeatherData有新的测量，这些布告必须马上更新。
- 此系统必须可扩展，让其他开发人员建立定制的布告板，用户可以随心所欲地添加或删除任何布告板。目前初始的布告板有三类：“目前状况”布告、“气象统计”布告、“天气预报”布告。

getTemperature()
getHumidity()
getPressure()

measurementsChanged()

将来的布告板

第一号布告板

第二号布告板

第三号布告板

先看一个错误示范

这是第一个可能的实现：我们依照Weather-O-Rama气象站开发人员的暗示，在measurementsChanged()方法中添加我们的代码：

```
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。

我们的实现有什么不对？

回想第 1 章的概念和原则……

```
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。

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。

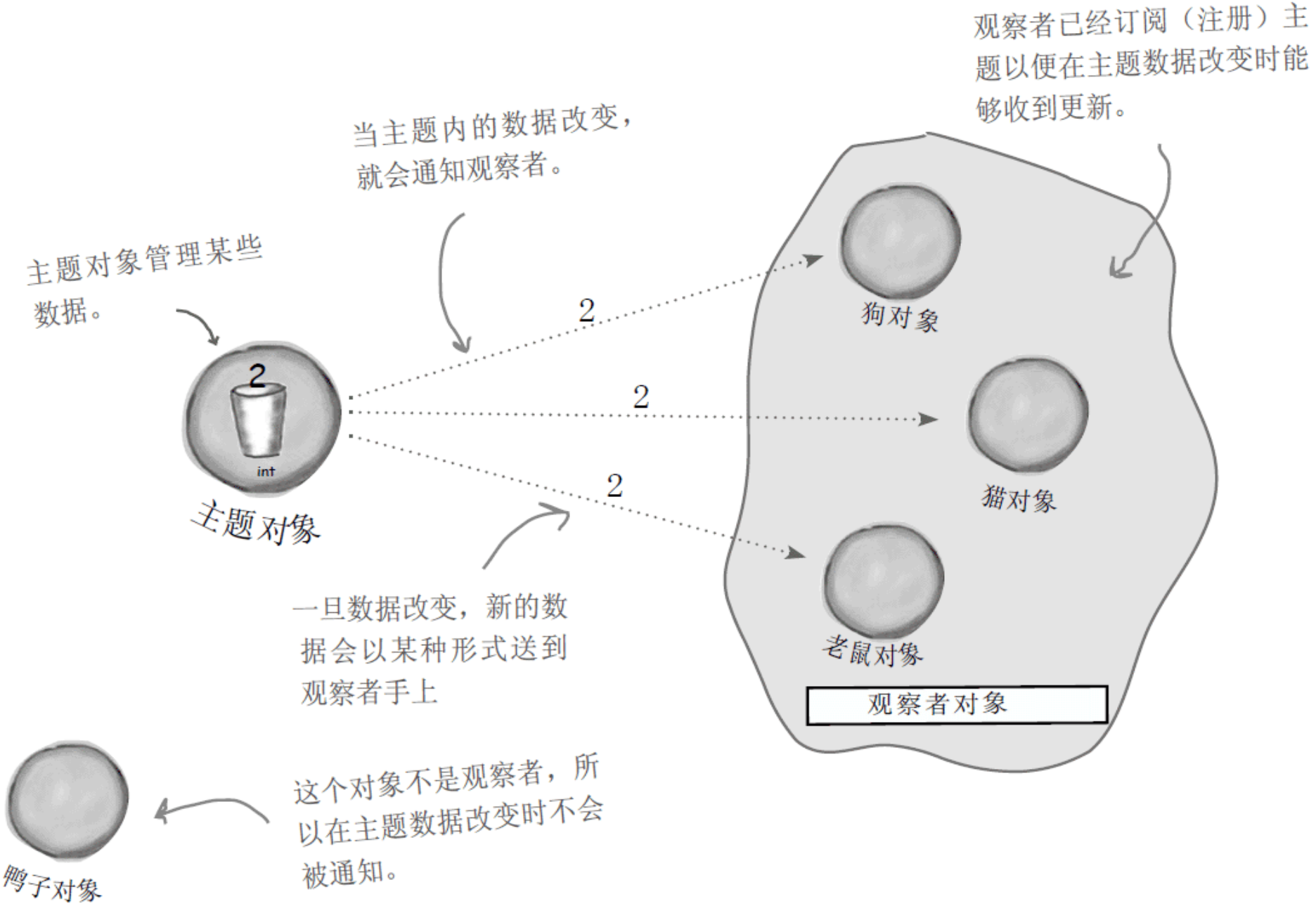
针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序。

认识观察者模式

- 我们看看报纸和杂志的订阅是怎么回事：
  - ① 报社的业务就是出版报纸。
  - ② 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。
  - ③ 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。
  - ④ 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。

出版者+订阅者=观察者模式

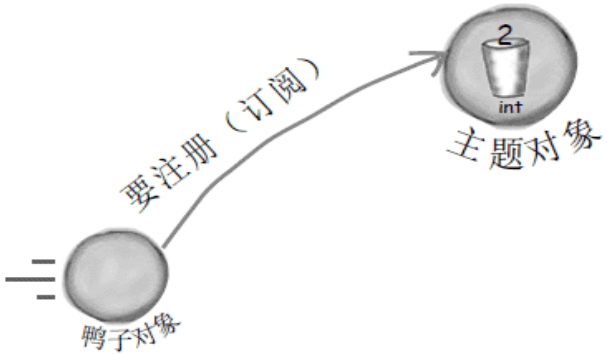
- 如果你了解报纸的订阅是怎么回事，其实就知道观察者模式是怎么回事，只是名称不太一样：出版者改称为“主题”（Subject），订阅者改称为“观察者”（Observer）。

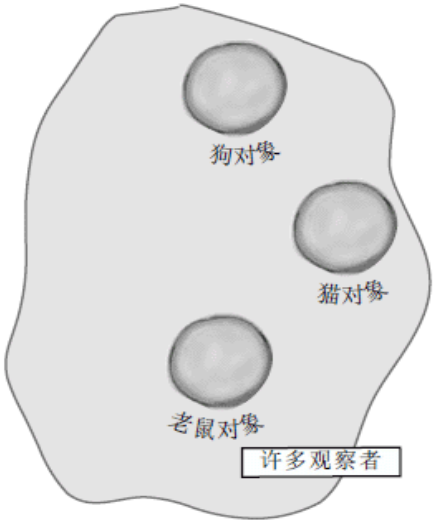

观察者已经订阅（注册）主题以便在主题数据改变时能够收到更新。
当主题内的数据改变，就会通知观察者。
主题对象管理某些数据。
2
int
主题对象
2
2
2
狗对象
猫对象
老鼠对象
观察者对象
一旦数据改变，新的数据会以某种形式送到观察者手上
这个对象不是观察者，所以在主题数据改变时不会被通知。
鸭子对象

观察者模式的一天

鸭子对象过来告诉主题，它想当一个观察者。

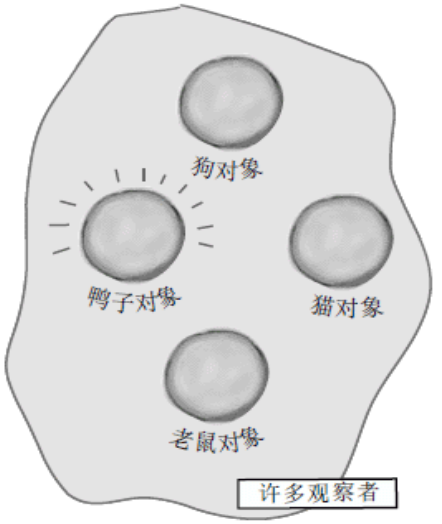
鸭子其实想说的是：我对你的数据改变感兴趣，一有变化请通知我





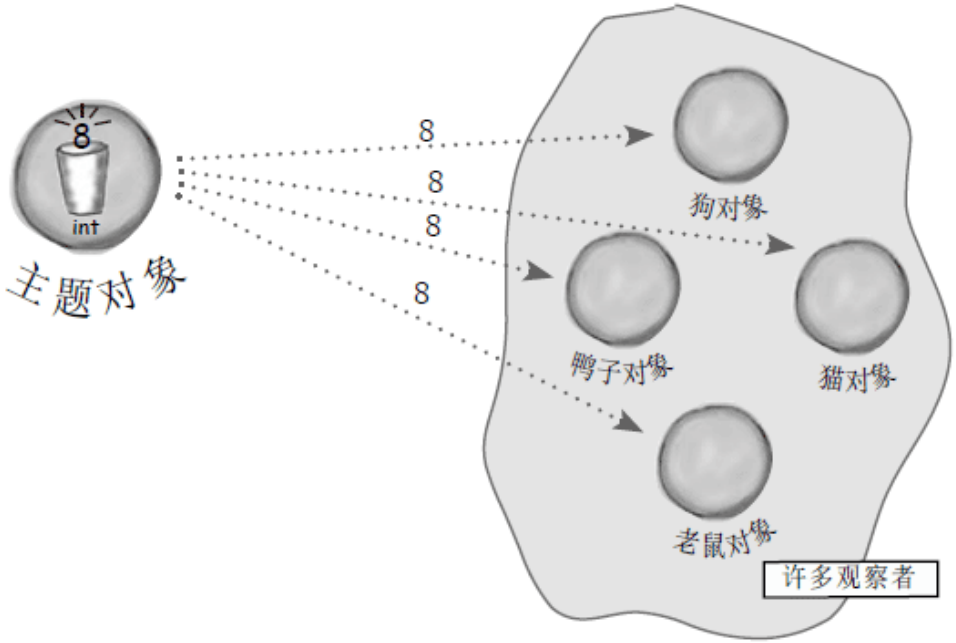
鸭子对象现在已经是正式的观察者了。

鸭子静候通知，等待参与这项伟大的事情。一旦接获通知，就会得到一个整数。



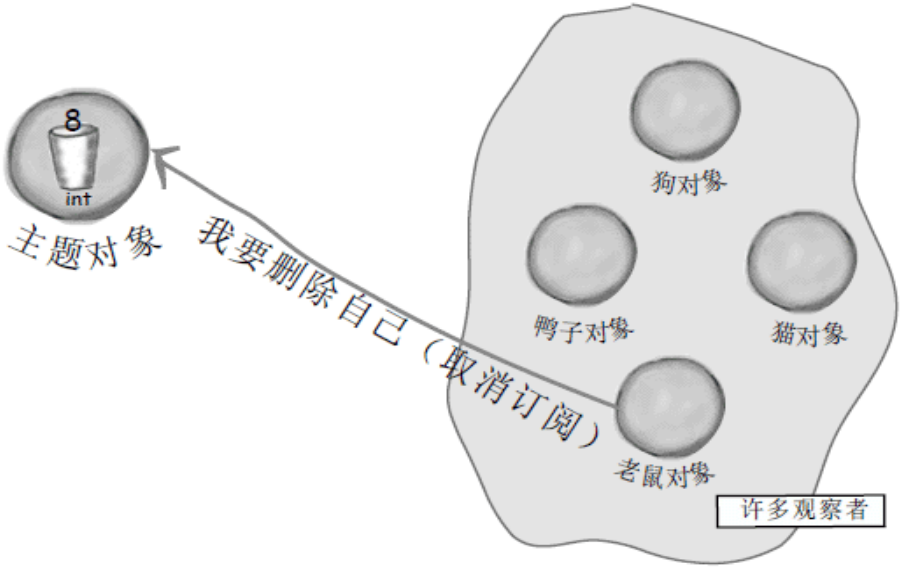
主题有了新的数据值！

现在鸭子和其他所有观察者都会收到通知：主题已经改变了。



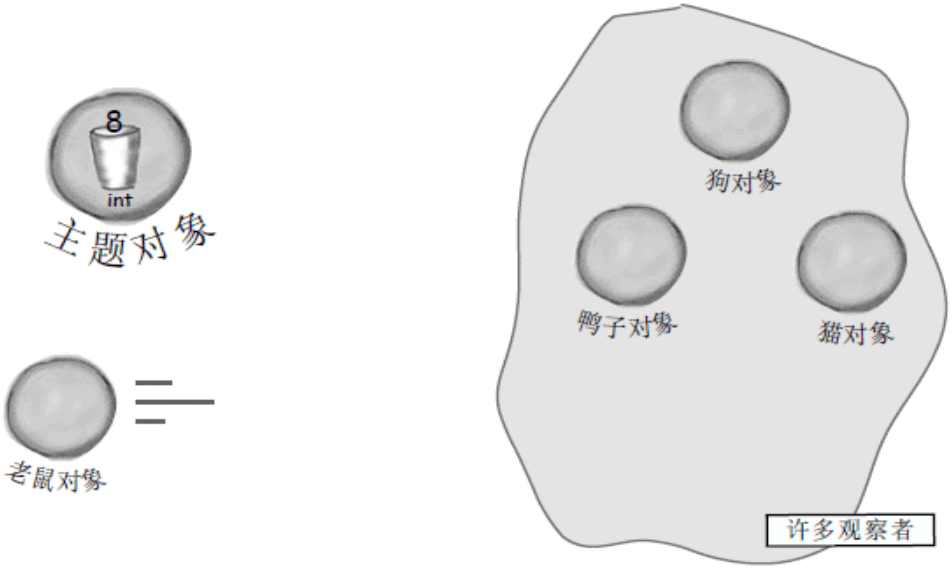
老鼠对象要求从观察者中把自己除名。

老鼠已经观察此主题太久，厌倦了，所以决定不再当个观察者。



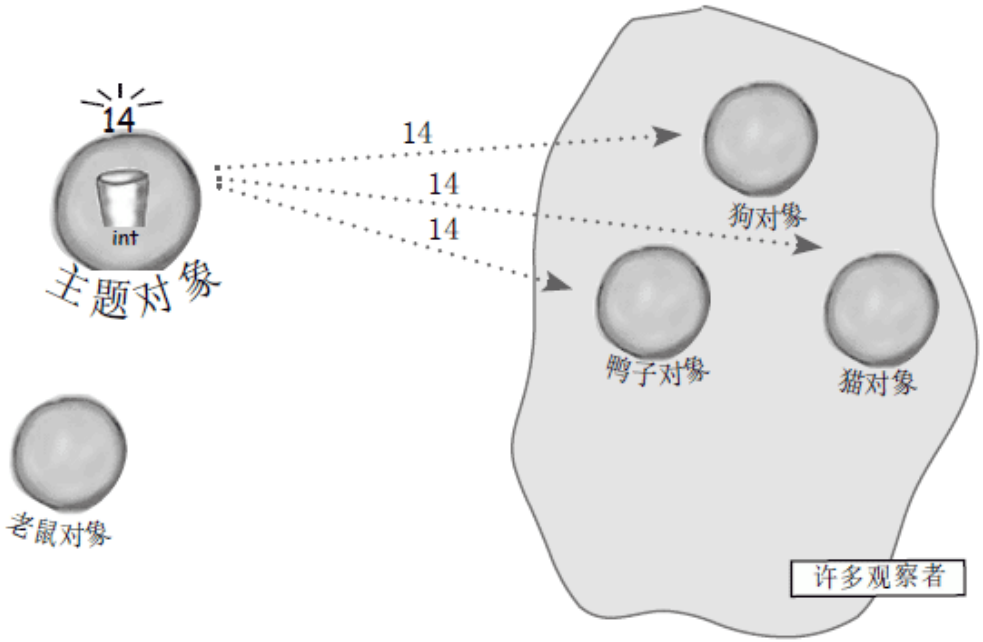
老鼠离开了！

主题知道老鼠的请求之后，把它从观察者中除名。



主题有一个新的整数

除了老鼠之外，每个观察者都会收到通知，因为它已经被除名了。嘘！不要告诉别人，老鼠其实心中暗暗地怀念这些整数，或许哪天又会再次注册，回来继续当观察者呢！



定义观察者模式

当你试图勾勒观察者模式时，可以利用报纸订阅服务，以及出版者和订阅者比拟这一切。

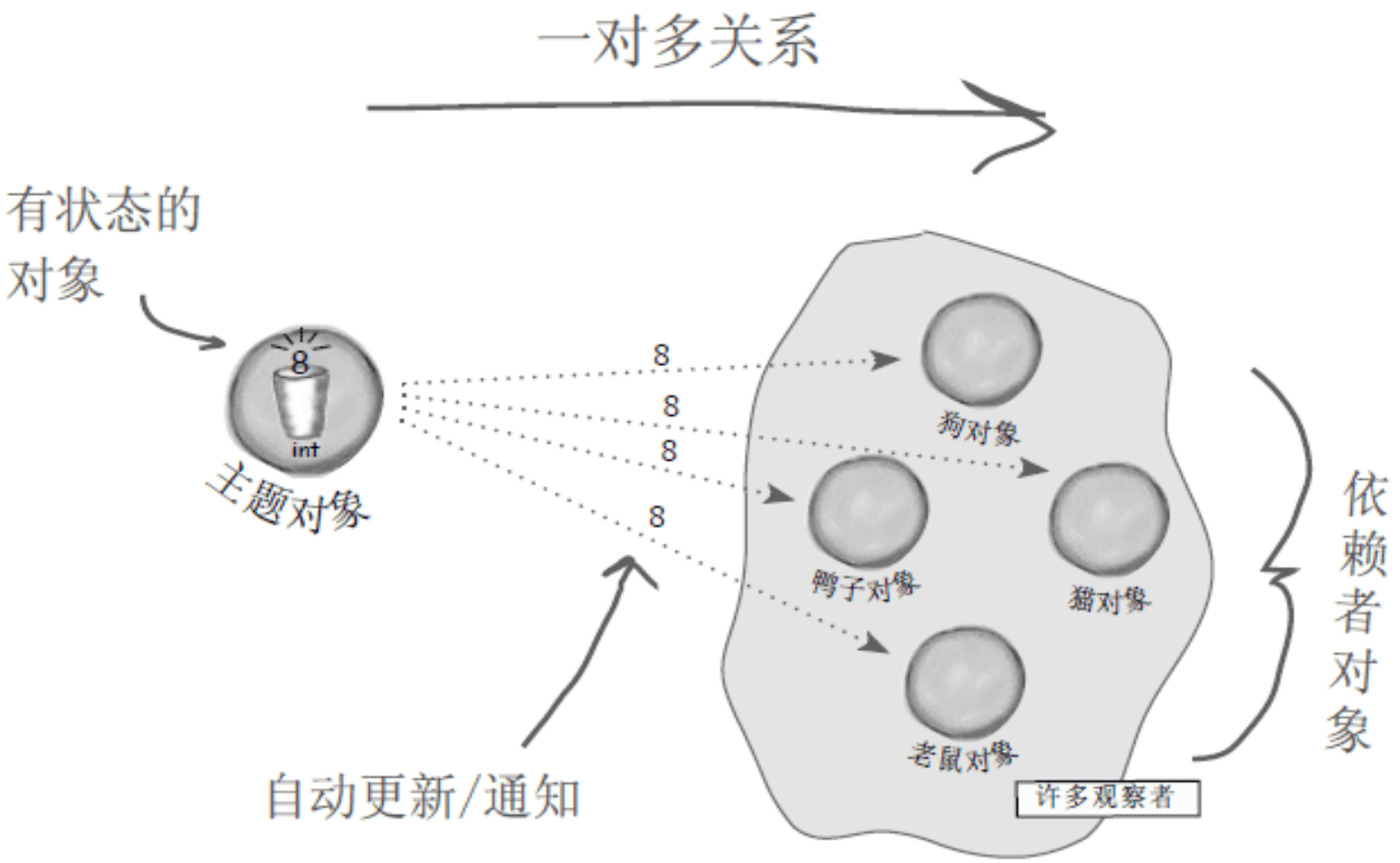
在真实的世界中，你通常会看到观察者模式被定义成：

> 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。



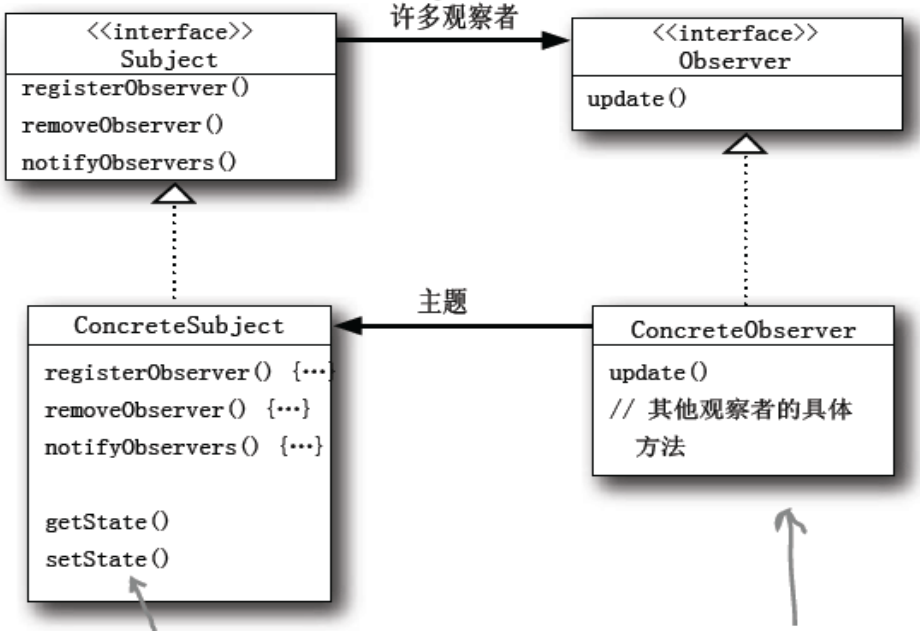
主题和观察者定义了一对多的关系。观察者依赖于此主题，只要主题状态一有变化，观察者就会被通知。根据通知的风格，观察者可能因此新值而更新。

定义观察者模式：类图

这是主题接口，对象使用此接口注册为观察者，或者把自己从观察者中删除。

每个主题可以有许多观察者

所有潜在的观察者必须实现观察者接口，这个接口只有update()一个方法，当主题状态改变时它被调用。



一个具体主题总是实现主题接口，除了注册和撤销方法之外，具体主题还实现了notifyObservers()方法，此方法用于在状态改变时更新所有当前观察者。

具体主题也可能有设置和获取状态的方法（稍后会进一步讨论）。

具体的观察者可以是实现此接口的任意类。观察者必须注册具体主题，以便接收更新。

there are no Dumb Questions

问： 这和一对多的关系有何关联？

答： 利用观察者模式，主题是具有状态的对象，并且可以控制这些状态。也就是说，有“一个”具有状态的主题。另一方面，观察者使用这些状态，虽然这些状态并不属于他们。有许多的观察者，依赖主题来告诉他们状态何时改变了。这就产生一个关系：“一个”主题对“多个”观察者的关系。

问： 其间的依赖是如何产生的？

答： 因为主题是真正拥有数据的人，观察者是主题的依赖者，在数据变化时更新，这样比起让许多对象控制同一份数据来，可以得到更干净的的OO设计。

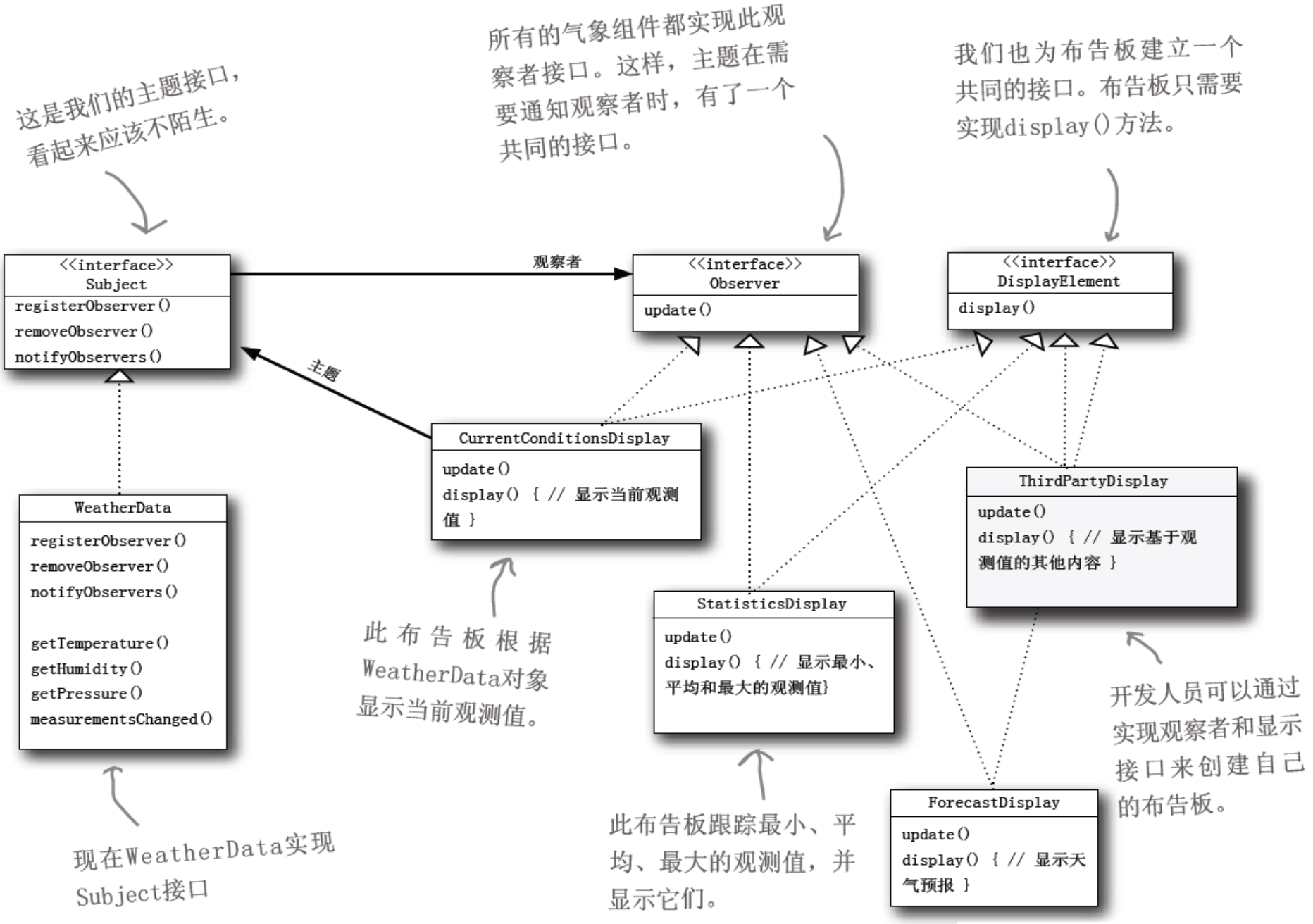
松耦合的威力

- 当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。

设计原则

为了交互对象之间的松耦合设计而努力。

松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的互相依赖降到了最低。


这是我们的主题接口，看起来应该不陌生。
所有的气象组件都实现此观察者接口。这样，主题在需要通知观察者时，有了一个共同的接口。
我们也为布告板建立一个共同的接口。布告板只需要实现display()方法。
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
观察者
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
主题
CurrentConditionsDisplay
update()
display() { // 显示当前观测值 }
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
此布告板根据WeatherData对象显示当前观测值。
StatisticsDisplay
update()
display() { // 显示最小、平均和最大的观测值}
ThirdPartyDisplay
update()
display() { // 显示基于观测值的其他内容 }
开发人员可以通过实现观察者和显示接口来创建自己的布告板。
ForecastDisplay
update()
display() { // 显示天气预报 }
现在WeatherData实现Subject接口
此布告板跟踪最小、平均、最大的观测值，并显示它们。

实现气象站

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

这两个方法都需要一个观察者作为变量，该观察者是用来注册或被删除的。

当主题状态改变时，这个方法会被调用，以通知所有的观察者。

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

当气象观测值改变时，主题会把这些状态值当作方法的参数，传送给观察者。

所有的观察者都必须实现update()方法，以实现观察者接口。在这里，我们按照Mary和Sue的想法把观测值传入观察者中。

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

DisplayElement接口只包含了一个方法，也就是display()。当布告板需要显示时，调用此方法。

在WeatherData中实现主题接口

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

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

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

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

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

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

WeatherData现在实现了Subject接口。

我们加上一个ArrayList来纪录观察者，此ArrayList是在构造器中建立的。

当注册观察者时，我们只要把它加到ArrayList的后面即可。

同样地，当观察者想取消注册，我们把它从ArrayList中删除即可。

这部分是Subject接口的实现。

有趣的地方来了！在这里，我们把状态告诉每一个观察者。因为观察者都实现了update()，所以我们知道如何通知它们。

当从气象站得到更新观测值时，我们通知观察者。

这部分是Subject接口的实现。

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

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

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.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();
    }

    // WeatherData的其他方法
}
```

当注册观察者时，我们只要把它加到ArrayList的后面即可。

同样地，当观察者想取消注册，我们把它从ArrayList中删除即可。

有趣的地方来了！在这里，我们把状态告诉每一个观察者。因为观察者都实现了update()，所以我们知道如何通知它们。

当从气象站得到更新观测值时，我们通知观察者。

我们想要每本书随书赠送一个小型气象站，但是出版社不肯。所以，和从装置中读取实际的气象数据相比，我们宁愿利用这个方法来测试布告板。或者，为了好玩，你也可以写代码从网站上抓取观测值。

我们已经把WeatherData类写出来了，现在轮到布告板了。Weather-O-Rama气象站订购了三个布告板：目前状况布告板、统计布告板和预测布告板。我们先看看目前状况布告板。一旦你熟悉此布告板之后，可以在本书的代码目录中，找到另外两个布告板的源代码，你会觉得这些布告板都很类似。

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);
    }

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

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

构造器需要 weatherData对象（也就是主题）作为注册之用。

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

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);
    }
}
```

首先，建立一个WeatherData对象。

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。

建立三个布告板，并把WeatherData对象传给它们。

模拟新的气象测量。

❷ 运行程序，让观察者模式表演魔术。

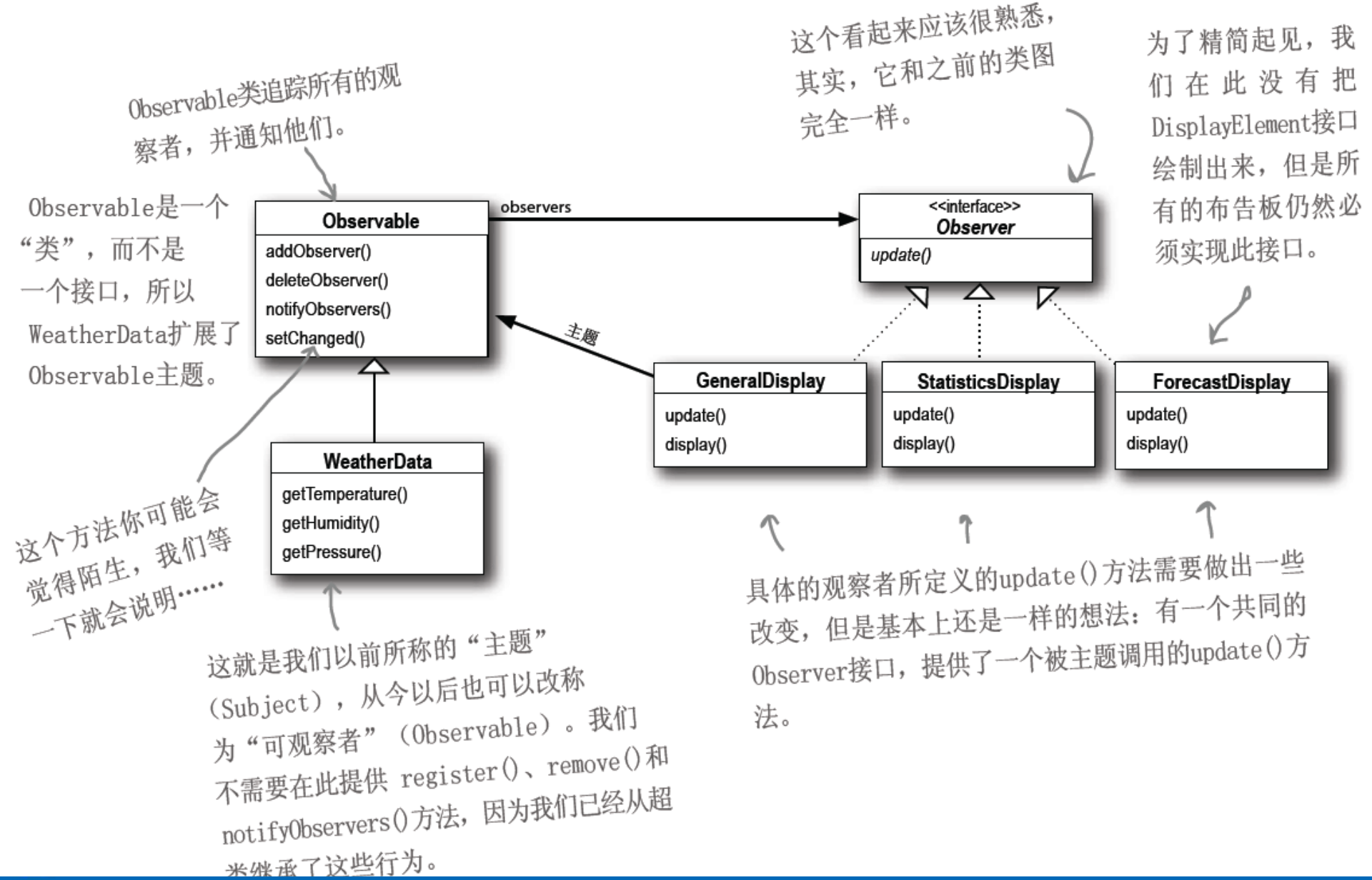
File Edit Window Help StormyWeather

```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

使用Java内置的观察者模式

到目前为止，我们已经从无到有地完成了观察者模式，但是，Java API有内置的观察者模式。java.util包（package）内包含最基本的Observer接口与Observable类，这和我们的Subject接口与Observer接口很相似。Observer接口与Observable类使用上更方便，因为许多功能都已经事先准备好了。你甚至可以使用推（push）或拉（pull）的方式传送数据，稍后就会看到这样的例子。

为了更了解java.uitl.Observer和java.util.Observable，看看下面的图，这是修改后的气象站OO设计。


Observable类追踪所有的观察者，并通知他们。
这个看起来应该很熟悉，其实，它和之前的类图完全一样。
为了精简起见，我们在此没有把DisplayElement接口绘制出来，但是所有的布告板仍然必须实现此接口。
Observable是一个“类”，而不是一个接口，所以WeatherData扩展了Observable主题。
Observable
addObserver()
deleteObserver()
notifyObservers()
setChanged()
observers
<<interface>>
Observer
update()
主题
GeneralDisplay
update()
display()
StatisticsDisplay
update()
display()
ForecastDisplay
update()
display()
WeatherData
getTemperature()
getHumidity()
getPressure()
这个方法你可能会觉得陌生，我们等一下就会说明……
这就是我们以前所称的“主题”（Subject），从今以后也可以改称为“可观察者”（Observable）。我们不需要在此提供 register()、remove()和notifyObservers()方法，因为我们已经从超类继承了这些行为。
具体的观察者所定义的update()方法需要做出一些改变，但是基本上还是一样的想法：有一个共同的Observer接口，提供了一个被主题调用的update()方法。

Java内置的观察者模式如何运作

Java内置的观察者模式运作方式，和我们在气象站中的实现类似，但有一些小差异。最明显的差异是WeatherData（也就是我们的主题）现在扩展自Observable类，并继承到一些增加、删除、通知观察者的方法（以及其他的方法）。Java版本的用法如下：

如何把对象变成观察者……

如同以前一样，实现观察者接口（java.uitl.Observer），然后调用任何Observable对象的addObserver()方法。不想再当观察者时，调用deleteObserver()方法就可以了。

可观察者要如何送出通知……

首先，你需要利用扩展java.util.Observable接口产生“可观察者”类，然后，需要两个步骤：

1. 先调用setChanged()方法，标记状态已经改变的事实。
2. 然后调用两种notifyObservers()方法中的一个：

```
notifyObservers()   或   notifyObservers(Object arg)
```

当通知时，此版本可以传送任何的数据对象给每一个观察者。

首先，你需要利用扩展java.util.Observable接口产生“可观察者”类，然后，需要两个步骤：

1. 先调用setChanged()方法，标记状态已经改变的事实。
2. 然后调用两种notifyObservers()方法中的一个：

```
notifyObservers()   或   notifyObservers(Object arg)
```

当通知时，此版本可以传送任何的数据对象给每一个观察者。

观察者如何接收通知……

同以前一样，观察者实现了更新的方法，但是方法的签名不太一样：

data object

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

主题本身当作第一个变量，好让观察者知道是哪个主题通知它的。

这正是传入notifyObservers()的数据对象。如果没有说明则为空。

如果你想“推”（push）数据给观察者，你可以把数据当作数据对象传送给notifyObservers(arg)方法。否则，观察者就必须从可观察者对象中“拉”（pull）数据。如何拉数据？我们再做一遍气象站，你很快就会看到。

Observable类的伪代码

```
setChanged() {
    changed = true
}

notifyObservers(Object arg) {
    if (changed) {
        for every observer on the list {
            call update (this, arg)
        }
        changed = false
    }
}

notifyObservers() {
    notifyObservers(null)
}
```

setChanged()方法把changed标志设为true。

notifyObservers() 只会在changed标为“true”时通知观察者。

在通知观察者之后，把changed标志设回false。

这样做有其必要性。setChanged()方法可以让你在更新观察者时，有更多的弹性，你可以更适当地通知观察者。比方说，如果没有setChanged()方法，我们的气象站测量是如此敏锐，以致于温度计读数每十分之一度就会更新，这会造成WeatherData对象持续不断地通知观察者，我们并不希望看到这样的事情发生。如果我们希望半度以上才更新，就可以在温度差距到达半度时，调用setChanged()，进行有效的更新。

你也许不会经常用到此功能，但是把这样的功能准备好，当需要时马上就可以使用。总之，你需要调用setChanged()，以便通知开始运转。如果此功能在某些地方对你有帮助，你可能也需要clearChanged()方法，将changed状态设置回false。另外也有一个hasChanged()方法，告诉你changed标志的当前状态。

利用内置的支持重做气象站

首先，把WeatherData改成使用java.util.Observable

❶ 记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在继承Observable。

❸ 我们不再需要追踪观察者了，也不需要管理注册与删除（让超类代劳即可）。所以我们把注册、添加、通知的相关代码删除。

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

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    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();
```

❹ 我们的构造器不再需要为了记住观察者们而建立数据结构了。

★ 注意：我们没有调用notifyObservers()传送数据对象，这表示我们采用的做法是“拉”。

❺ 在调用notifuObservers()之前，要先调用setChanged()来指示状态已经改变。

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

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    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();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

4 我们的构造器不再需要为了记住观察者们而建立数据结构了。

★ 注意：我们没有调用notifyObservers()传送数据对象，这表示我们采用的做法是“拉”。

5 在调用notifuObservers()之前，要先调用setChanged()来指示状态已经改变。

6 这些并不是新方法，只是因为我们要使用“拉”的做法，所以才提醒你有这些方法。察者会利用这些方法取得WeatherData对象的状态。

1 再说一遍，记得要导入（import）正确的Observer/Observable。

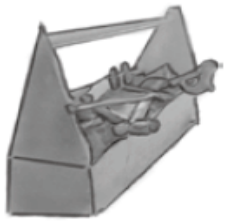
2 我们现在正在实现java.util.Observer接口。

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

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

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

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

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

3 现在构造器需要一Observable当参数，并将CurrentCondi-tionsDisplay对象登记成为观察者。

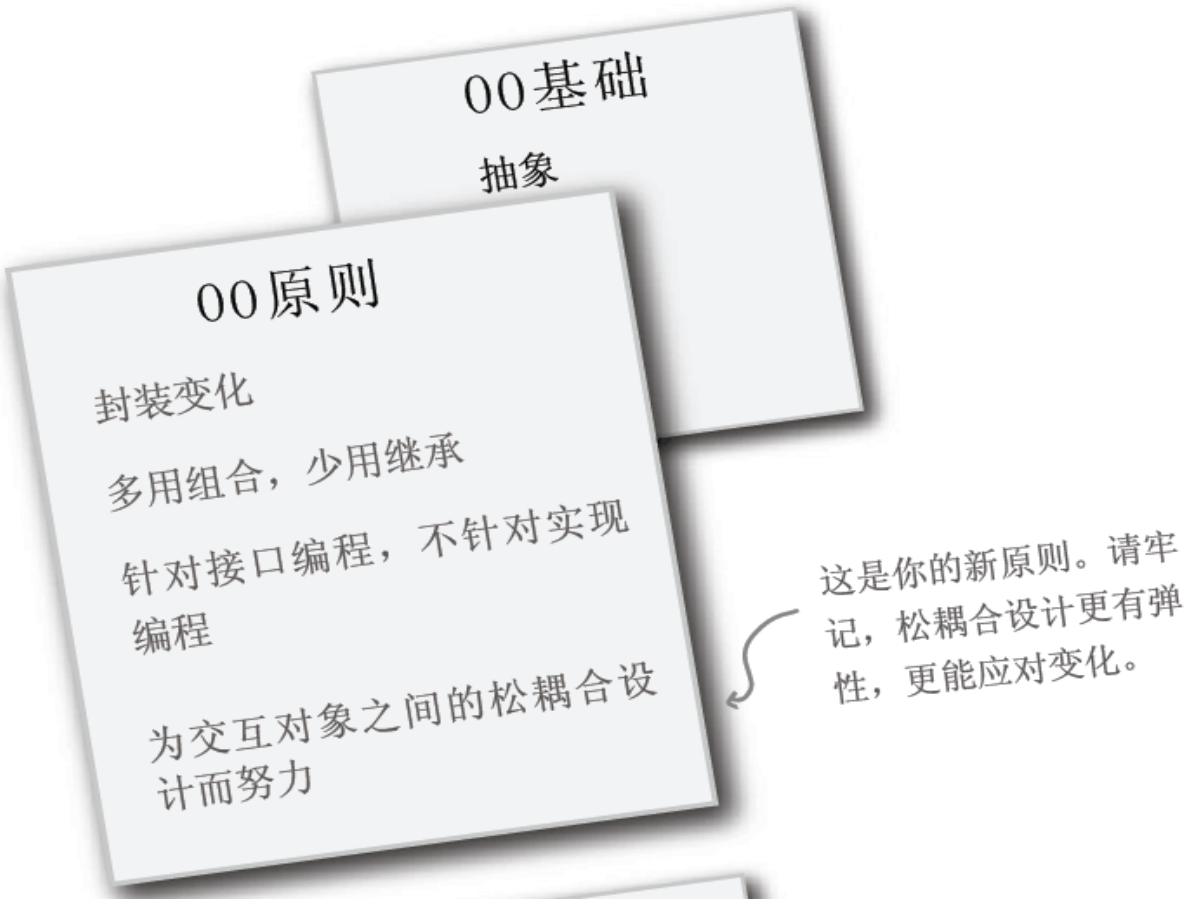
4 改变update()方法，增加Observable和数据对象作为参数。

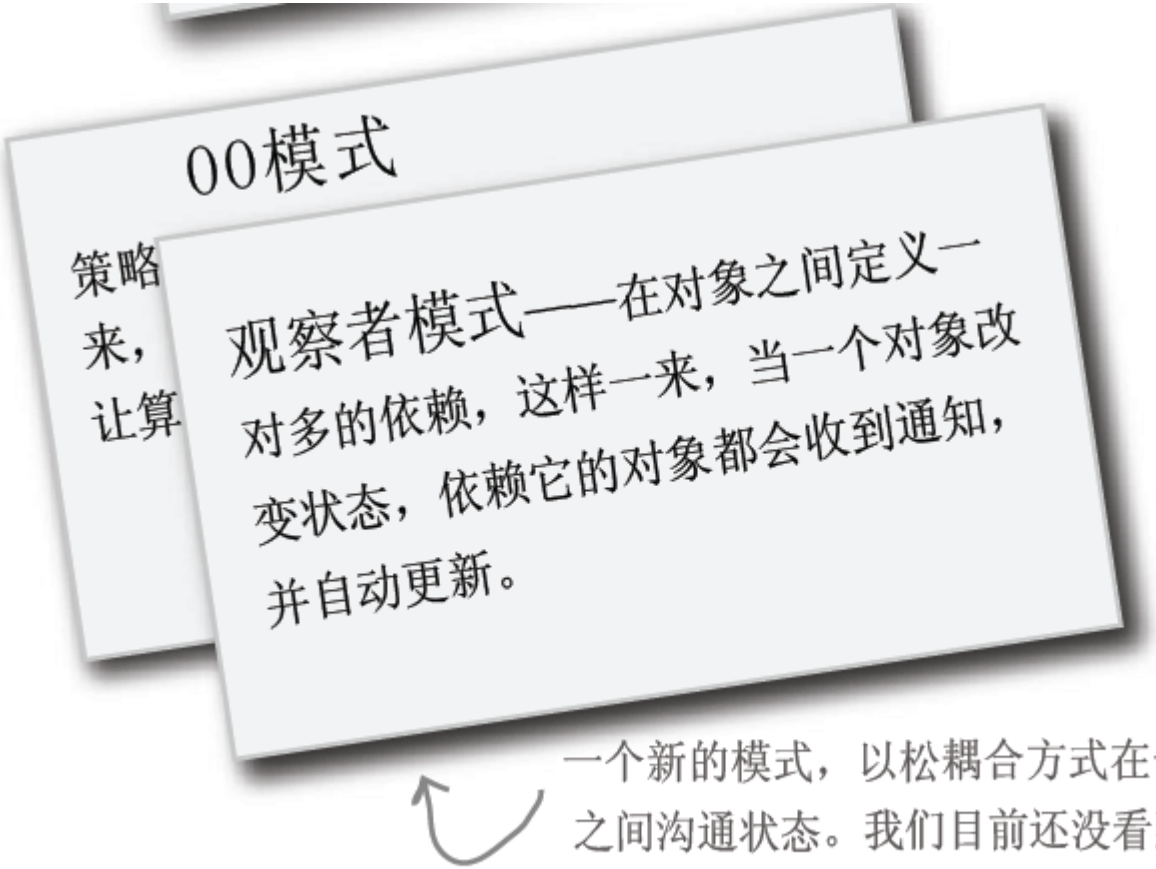
5 在 update()中，先确定可观察者属于WeatherData类型，然后利用 getter方法



设计箱内的工具

欢迎来到第2章的结尾，你的对象工具箱内又多了一些东西……





一个新的模式，以松耦合方式在一系列对象之间沟通状态。我们目前还没看到观察者模式的代表人物——MVC，以后就会看到了。