设计模式之观察者模式（Observer Pattern）

六尺帐篷 (/u/f8e9b1c246f1)
2016.07.23 16:55 字数 1895 阅读 474 评论 6 喜欢 16
(/u/f8e9b1c246f1)

编辑文章 (/writer#/notebooks/5229761/notes/4926532)

在正式介绍观察者模式前，我们先引用生活中的小例子来模拟观察者，先对观察者模式有一个整体的感觉。

现实模拟

报纸和杂志的故事。
我们看看报纸和杂志的订阅是怎么一回事：

- 报纸的任务就是出版报纸
- 我们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，你就会一直得到新报纸
- 当你们不想再看报纸的时候，向报社取消订阅，他们就不会再送报纸来，你也不会再收到报纸
- 只要报社还在运营，就会有人向他们订阅或者取消报纸

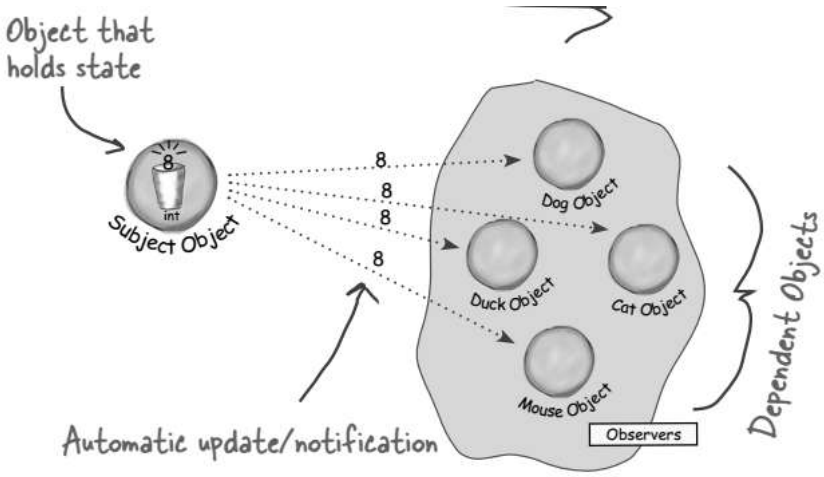
这其实就可以理解为是一种观察者模式。报社出版者被认为是观察者模式中的**Subject**，订阅报纸的人被认为是观察者模式中的**Observer**。具体的观察者模式的subject和observer我们后面会介绍。

订阅者通常有很多个，他们订阅或者取消需要通知出版者。出版者当报纸有更新时，就会把新报纸一起推送给订阅者，所有订阅者都会收到出版社的所有更新。

再举个常见的例子，我们常见的手机app，网易新闻或者其他类。只要我们安装了这个这个应用，并在app设置接收应用的消息通知，那么当app有新消息通知时，我们就会收到新消息。这里，我们用户就是观察者，app就是Subject。

观察者模式定义

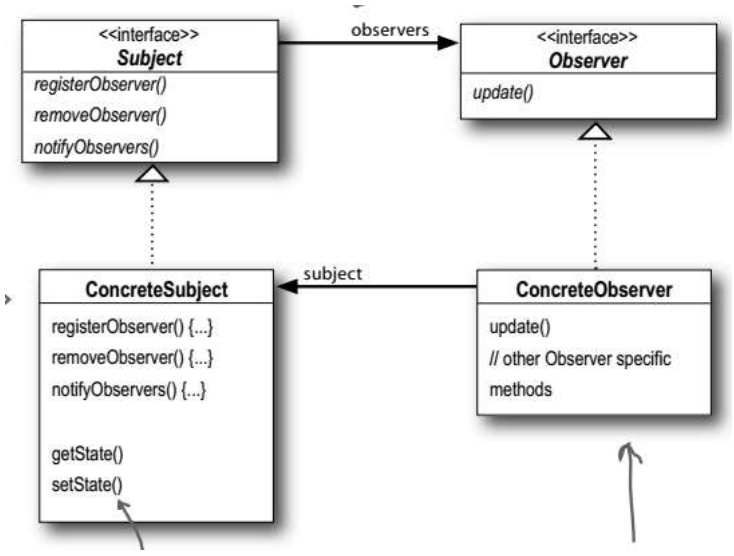
观察者模式是设计模式中很常用的一个模式。
比较严格的解释是： **观察者模式定义了对象之间的一对多的依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。**



Paste_Image.png

跟图中的例子一样，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。

观察者模式的类图可以很好的观察者模式的设计思想



Paste_Image.png

观察者的设计方式有很多种，但其中实现Subject和observer接口的设计方式是最常用的、

Subject的接口有三个方法，分别是注册观察者，移除观察者和通知观察者。对象通过Subject接口注册成为观察者，同事也可以通过它从解除观察者的身份，也就是之前例子中的取消订阅报纸。

每个Subject通常可以有很多个观察者

具体的Subject对象需要实现Subject接口的三个方法，其中notify方法是用于当状态发生变化时，来通知观察者update，里面一般要调用观察者接口的update方法。

所有的观察者都需要实现Observer接口，并实现其中的update方法，以便当主题状态发生变化，观察者得到主题的通知。用于Subject具体实现类的notify方法的调用。

具体的Observer都需要继承至接口，同时他们必须注册到具体的Subject对象，以成为一个观察者，并得到更新。

观察者实现的设计原则

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，主题只需要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的实现类是谁，做了些什么或者其他任何细节，主题都不需要知道。

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，主题不会受任何影响。同样的，也可以在任何时候删除观察者。

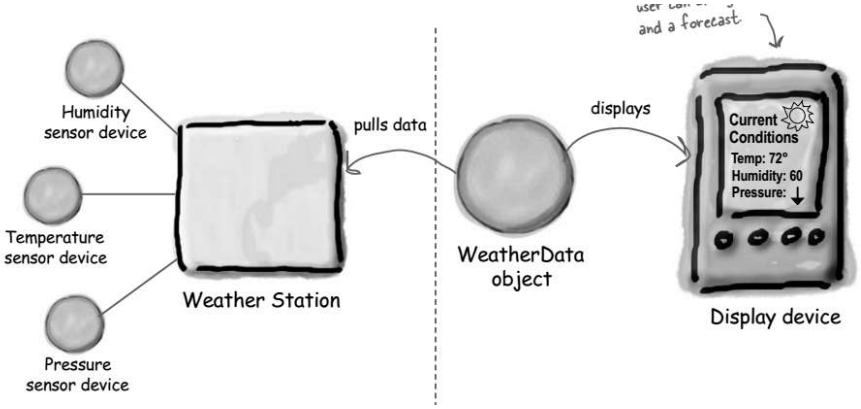
当有新的类型的观察者出现时，主题的代码不会发生修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所需要的只是在新的类里实现此观察者的接口，然后注册为观察者即可。

这里体现了一个设计原则就是 **为了交互对象之间的松耦合设计而努力**
争取让对象之间的互相依赖降到最低

代码实现

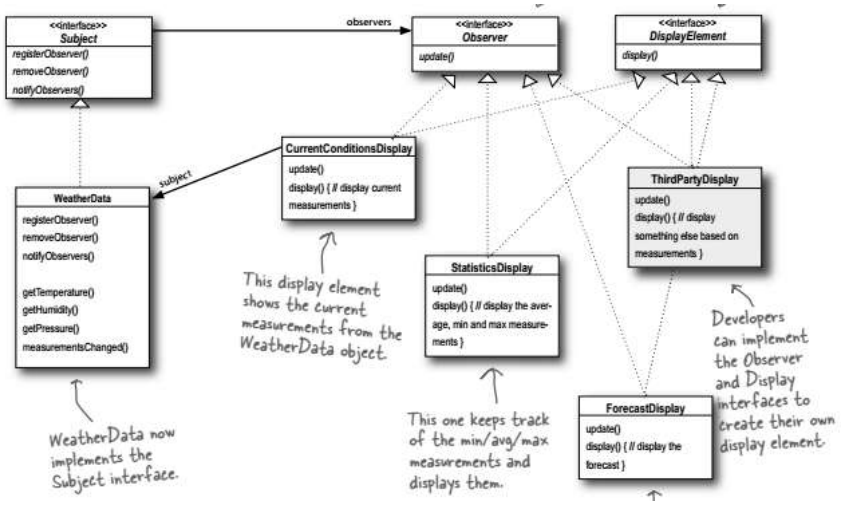
我们考虑这样一个问题：实现一个气象站监测应用。

有三个部分，气象站（获取实际气象数据的装置），weatherData对象（追踪来自气象站的数据，并更新布告板）和布告板（显示目前天气的状况给用户看）



Paste_Image.png

我们要做到的就是建立一个应用，利用weatherdata对象获取数据，并更新三个布告板。
我们对气象站的初步设计图：



Paste_Image.png

根据观察者设计了一个类图，接下来我们实现这个类图。从建立接口开始，

```
package com.liu.itf;

public interface Subject {
    public void registerObserver(Observers o);
    public void removeObserver(Observers o);
    public void notifyObserver();
}
```

```
package com.liu.itf;

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

```
package com.liu.itf;

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

接下来在weatherdata类中实现Subject接口

```
package com.liu.model;

import java.util.ArrayList;

import com.liu.itf.Observers;
import com.liu.itf.Subject;

public class WeatherData implements Subject {

    private ArrayList<Observers> observers;
    private float temperature;
    private float pressure;
    private float humidity;

    public WeatherData() {
        // TODO Auto-generated constructor stub
        observers = new ArrayList<Observers>();
    }

    @Override
    public void registerObserver(Observers o) {
        // TODO Auto-generated method stub
        observers.add(o);
    }

    @Override
    public void removeObserver(Observers o) {
        // TODO Auto-generated method stub
        int i = observers.indexOf(o);
        if(i>=0) {
            observers.remove(o);
        }
    }

    @Override
    public void notifyObserver() {
        // TODO Auto-generated method stub
        for(int i=0;i<observers.size();i++) {
            Observers observer = (Observers)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObserver();
    }

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

}
```

布告板类作为观察者实现观察者接口和display接口

```
package com.liu.view;

import com.liu.itf.DisplayElement;
import com.liu.itf.Observers;
import com.liu.itf.Subject;

public class CurrentConditionDisplay implements DisplayElement, Observers {

    private float temperature;
    private float pressure;
    private float humidity;
    private Subject weatherData;

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

    @Override
    public void update(float temp, float humidity, float pressure) {
        // TODO Auto-generated method stub
        this.temperature = temp;
        this.humidity = humidity;
        display();
    }

    @Override
    public void display() {
        // TODO Auto-generated method stub
        System.out.println("Current conditions: " + temperature + "F degrees and " + humidit
    }

}
```

编写一个测试类：

```
import com.liu.model.WeatherData;
import com.liu.view.CurrentConditionDisplay;


public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionDisplay currentConditionDisplay = new CurrentConditionDisplay(weathe

        weatherData.setMeasurements(80, 45, 30.4f);
    }
}
```

小结

- 观察者定义了对象之间一对多的关系。
- 主题用一个共同的接口来更新观察者
- 观察者和主题之间用松耦合的方式连接，主题不知道观察者的细节，只知道观察者实现了观察者接口

设计模式DesignPattern (/nb/5229761)

六尺帐篷 (/u/f8e9b1c246f1)
写了 245418 字，被 15240 人关注，获得了 1429 个喜欢
(/u/f8e9b1c246f1)