SOFTWARE ENGINEERING INSTITUTE
华东师范大学软件学院

Object oriented Analysis &Design
面向对象分析与设计

Lecture_16 Design Pattern: Observer

主讲: 陈小红

日期:

GoF设计模式的分类

（1）Creational（创建型）5个

（2）Structural（结构型）7个

（3）Behavioral（行为型）11个

	创建型	结构型	行为型
类	Factory Method	Adapter_Class	Interpreter Template Method
对象	Abstract Factory Builder Prototype Singleton	Adapter_Object Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

现实场景1-微信公众号


中国联通 下午6:42 44%
微信
订阅号
新榜
融资首发 | 两年估值涨10倍，这个女性新媒...
昨天
新榜排行榜
网易号9月财经科技数码榜单 | 多元内容绽放...
昨天
媒记
熬夜修仙、游戏成瘾、酒精中毒..."作死"成...
2017/9/15
新华网
[2条] 进入新时代！新华社献上了这首歌，据...
下午6:40
有车以后
[19条] 爆料！有车以后的"神秘"人物，终于...
下午6:34
环球时报
[8条] 十九大报告让岛内悟到：现在统一台湾...
下午6:32
人民日报
[34条] 这些你最关心的问题，十九大报告都...
下午6:22
张佳玮写字的地方
[2条] 理想中，2018年世界杯应该是怎么样...
下午6:03
观察者网
[24条] 厉害了...原来发微博报警真的有用？！
下午6:00
中国联通 下午6:42 43%
订阅号
新榜
微信大号"学生时代"被封，运营者这样回应（图）| 新榜情报
昨天 下午6:45
融资首发 | 两年估值涨10倍，这个女性新媒体为何再获3000万融资？
一个人写太难，几个人写不易，合作写稿怎么分工？风格如何一致？
有问有答
从招募拉群到课程管理，大V们的付费社群是怎么做起来的？
微信大号"学生时代"被封，运营者这样回应（图）| 新榜情报
做广告
卖东西
找新榜


微信 | 公众平台
二维码订阅
消息推送
品牌传播
更多...

现实场景2-电商降价通知

电商降价通知

- 送牛奶
- 订阅报纸

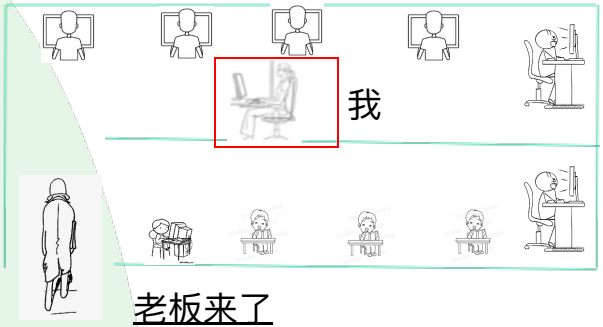
Observer

- A way of notifying change to a number of classes

老板来了，我要知道.

实验室的例子

- 我坐在实验室里



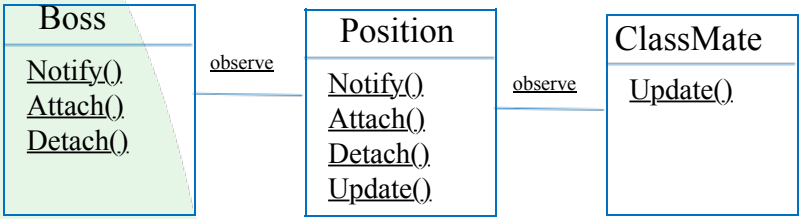
1.师兄弟姐妹告诉我老板来了，要通知他们，我说老板好的时候，大家马上切屏。

2. 师兄师姐们毕业了，新人进来了。

3. 我毕业了，新的师弟代替了我做这个工作。

4. 后来这个师弟又毕业了，又进了新人。

2次观察

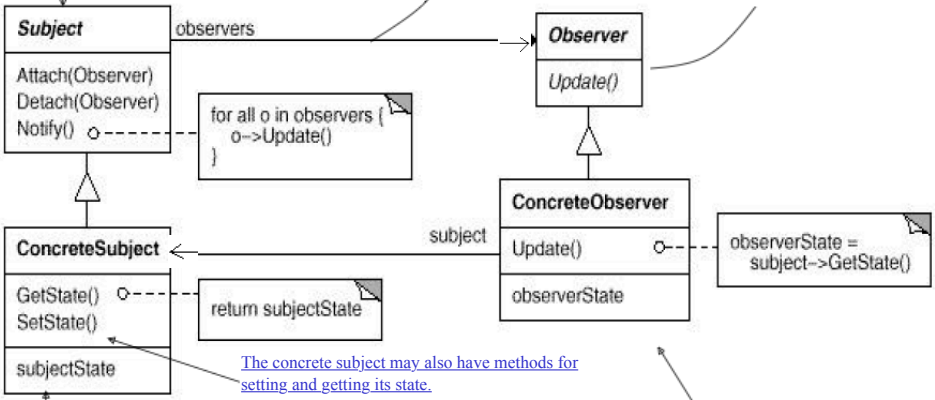

Boss
Notify()
Attach()
Detach()
observe
Position
Notify()
Attach()
Detach()
Update()
observe
ClassMate
Update()

Observer Class Diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers

All potential observers need to implement the Observer interface. This interface has just one method, update ( ), that gets called when the Subject's state changes.



The concrete subject may also have methods for setting and getting its state.

A concrete subject always implements the Subject interface. In addition to the register (attach) and remove (detach) methods, the concrete subject implements a notify() method to notify observers whenever state changes.

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

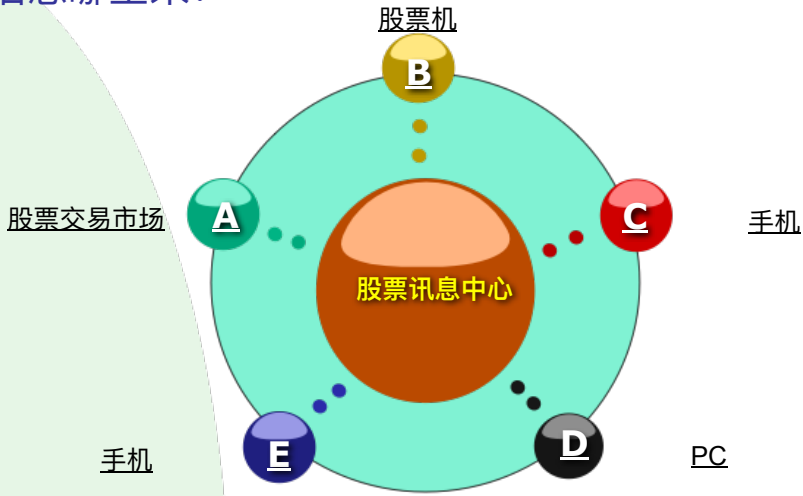
Observer Design pattern 观察者模式

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

定义对象之间的一对多依赖关系，当一个对象改变状态时，所有依赖于它的对象都会自动获得通知

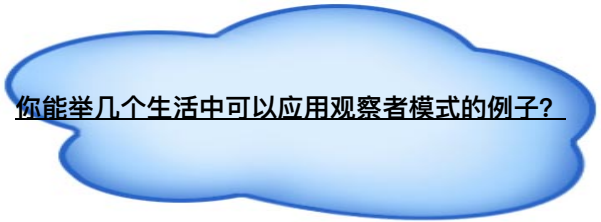
股票市场

信息哪里来?



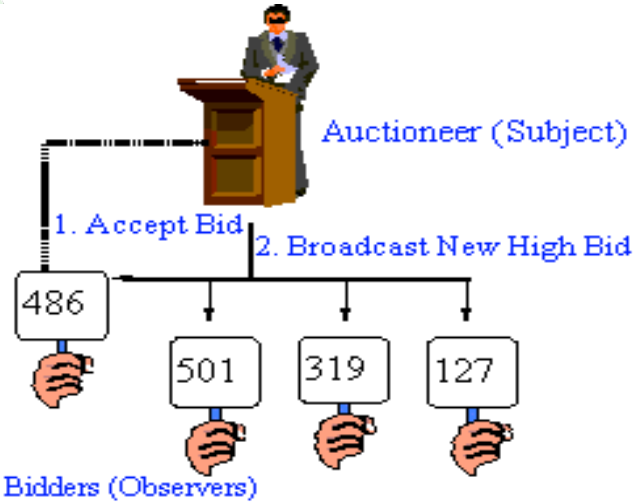
观察者(Observer)模式

- 上面介绍的就是一个典型的Observer设计模式。
- 观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式



Example

auctions



The Newspaper or Magazine subscription model

1. Publisher publish newspapers
2. You subscribe to a particular newspaper
3. You unsubscribe when you don't want the newspapers anymore
4. The other subscribers still get newspapers

What if the newspaper changes…

If publisher changes the subject of the newspaper,
Each subscriber (observer) is notified.
So they can take their own steps.

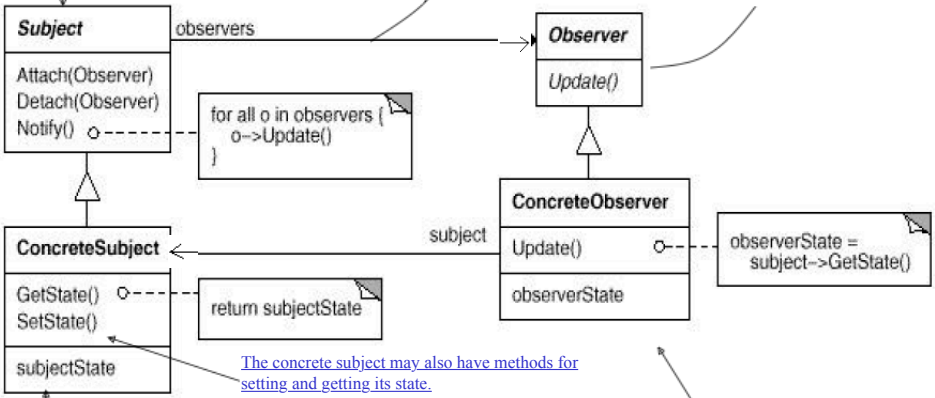
Observer Class Diagram

SOFTWARE ENGINEERING INSTITUTE
華東師范大學软件学院

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers

All potential observers need to implement the Observer interface. This interface has just one method, update ( ), that gets called when the Subject's state changes.

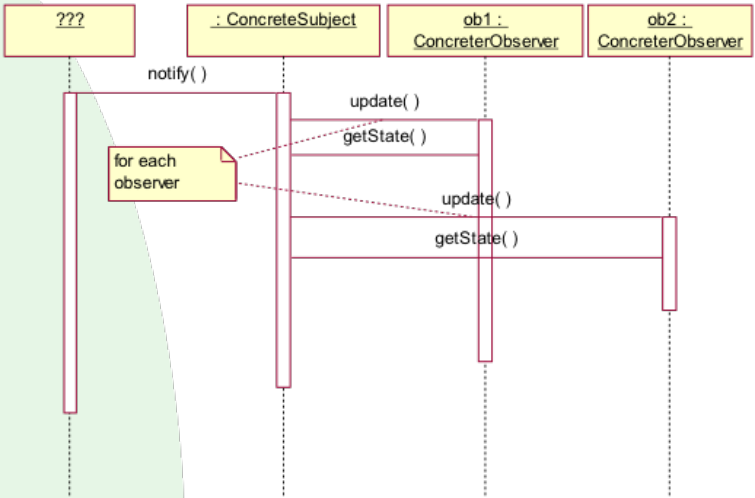


The concrete subject may also have methods for setting and getting its state.

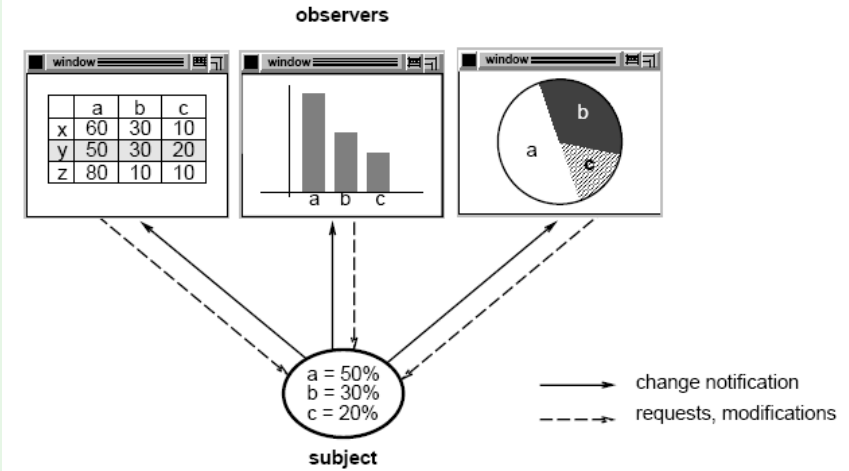
A concrete subject always implements the Subject interface. In addition to the register (attach) and remove (detach) methods, the concrete subject implements a notify() method to notify observers whenever state changes.

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Observer Pattern Sequence Diagram

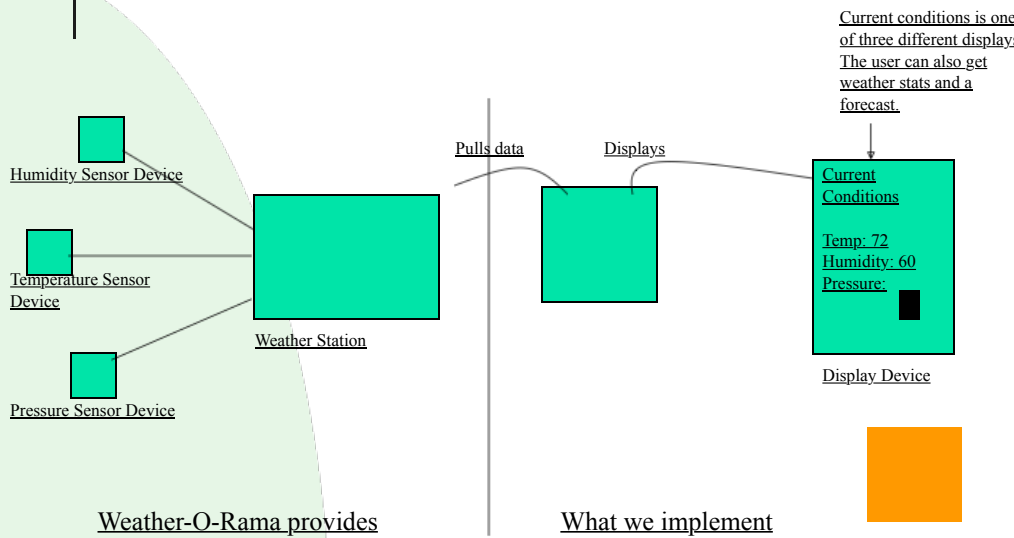


Subject-observer



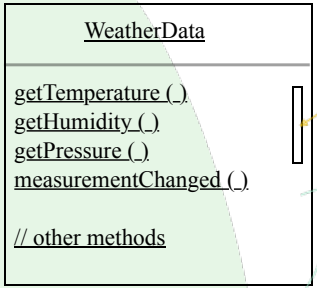
[from website

(Exam.)The Weather-O-Rama!(公司名)



The Job: Create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

The WeatherData class



These three methods return the most recent weather measurements for temperature, humidity, and pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated information from the Weather Station

A clue: what we need to add!

```
/*
 * This method gets called whenever the
 * measurements have been updated.
 */
public void measurementsChanged ( ){
        // Your code goes here
}
```

The Specs so far

- The WeatherData class has getter methods for three measurement values: temperature, humidity, and pressure.
- The measurementsChanged ( ) method is called anytime new weather measurement data is available.
  - We don't know or care how this method is called; we just know that it is
- We need to implement three display elements that use the weather data:
  - a *current conditions* display,
  - a *statistics* display,
  - and a *forecast* display.

  These displays must be updated each time WeatherData has new measurements.
- The system must be expandable
  - -- other developers can create new custom display elements and users can add or remove as many display elements as they want to the application.

A First Misguided Attempt at the Weather Station

```
public class WeatherData {
  // instance variable declarations
  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);
    }
    // other WeatherData methods here
}
```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented)

Now update the displays.

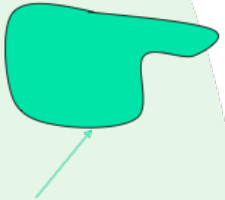
Call each display element to update its display, passing it the most recent measurements.

What's wrong with the first implementation?

```
public class WeatherData {
  // instance variable declarations
  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);
  }
  // other WeatherData methods here
}
```

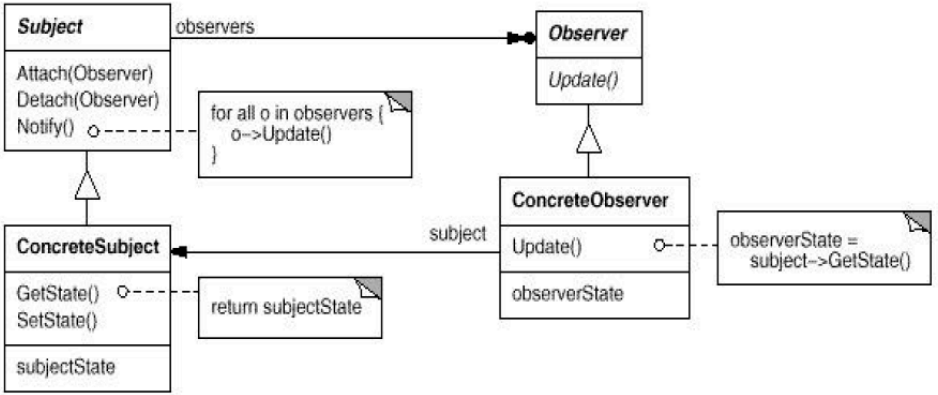
Area of change, we need to encapsulate this.



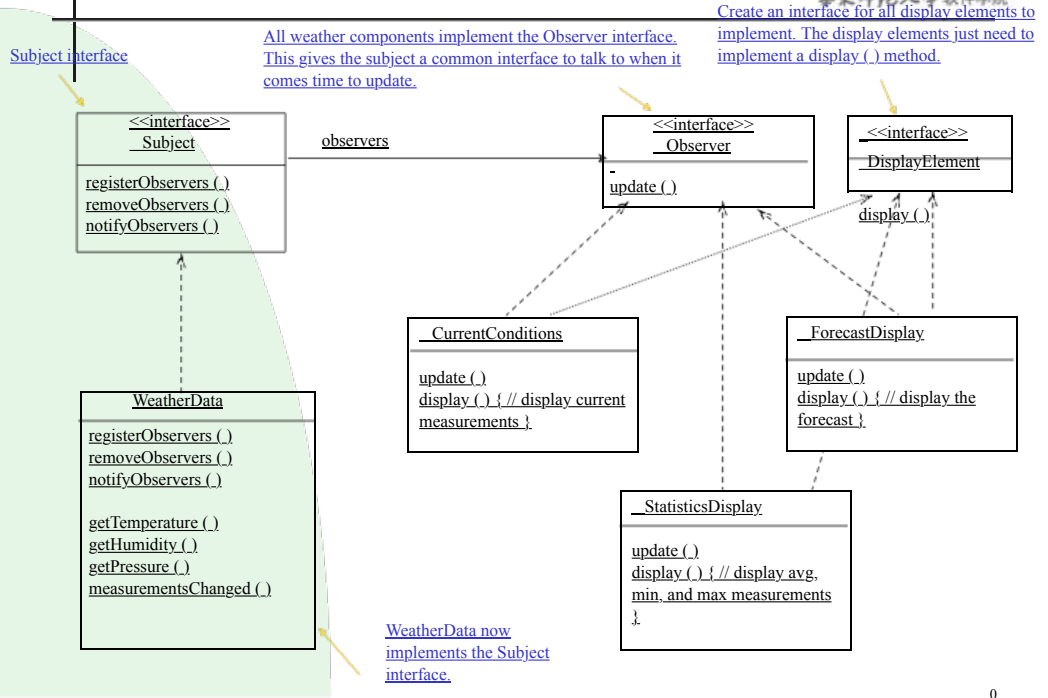
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

At least we seem to be using a common interface to talk to the display elements…they all have an update ( ) method that takes temp, humidity and pressure values.

It's Time for Observer



Designing the Weather Station



Implementing the Weather Station

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

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

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

Both of these methods take an Observer as an argument, that is the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

The Observer interface is implemented by all observers, so they all have to implement the update ( ) method.

These are the state values the Observers get from the Subject when a weather measurement changes.

The DisplayElement interface just includes one method, display ( ), that we will call when the display element needs to be displayed.

Implementing the Subject Interface in WeatherData


```
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 j = observer.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 ( ) {
```

Added an ArrayList to hold the Observers, and we create it in the constructor

Here we implement the Subject Interface

Notify the observers when measurements change.

The Display Elements

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
  private float temperatue;
  private float humidity;
  private Subject weatherData;

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

Implements the Observer and DisplayElement interfaces

The constructors passed the weatherData object (the subject) and we use it to register the display as an observer.

When update ( ) is called, we save the temp and humidity and call display ( )

The display ( ) method just prints out the most recent temp and humidity.

什么时候用观察者模式?

- 某一个对象的状态发生变化的时候，某些其它的对象需做出相应的改变 。
- 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。

- 观察者模式的两种形式
  - 推模式
    - —— 能把股票价格传过来的股票机
  - 拉模式
    - —— 只是起通知作用的股票机

推模式

- 推模式是当通知消息来之时，把所有相关信息都通过参数的形式“推给”观察者。

优点：

1. 所有信息通过参数传递过来，直接、简单，观察者可以马上进行处理。
2. 观察者与被观察者没有一点联系，两者几乎没有耦合

缺点：

1. 所有信息都强迫推给观察者，不管有用与否。
2. 如果想添加一个参数，那就需要修改所有观察者的接口函数。

拉模式

- 当通知消息来之时，通知的函数不带任何相关的信息，而是要观察者主动去被观察的对象那里去“拉”信息。

优点：

1. 可以主动去取自己感兴趣的信息。
2. 如要添加一个参数，无需修改观察者。

缺点：

1. 观察者与被观察者有一定的联系 。

【上海移动】11月流量账单已送达您的139邮箱！点击查看账单详情 http://y.10086.cn/t/fbQ2tGMtiNv21d 不限速网盘，http://wapmail.10086.cn/p/cy 回Q关闭通知【中国移动 139邮箱】

灵活增减观察者

- 可以灵活地增减观察者

当Roger不炒股，就去注销账号，他就不会再收到股票行情了，当然如果他以后还想炒股，那他再注册就行了。

对于股票信息中心，它是不知道每个投资者的具体情况的，它也无需关心这些，它仅仅知道的是观察者的接口（Update）。

这里就像订牛奶一样，牛奶公司只知道你的地址，把牛奶每天按时送到就可以了，而牛奶公司无需知道你家有几口人等信息。

How to apply Observer DP

- **Check list**
  - Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
  - Model the independent functionality with a "subject" abstraction.
  - Model the dependent functionality with an "observer" hierarchy.
  - The Subject is coupled only to the Observer base class.
  - The client configures the number and type of Observers.
  - Observers register themselves with the Subject.
  - The Subject broadcasts events to all registered Observers.
  - The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

Conclusion-Observer Pattern

- Define a one-to-many **dependency** between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Two types**: pull model** vs. **push model**

问题:

不同类型的订阅者对象关注于发布者对象的状态变化或事件,并且想要在发布者产生事件时以自己独特的方式作出反应.

解决方案:

定义订阅者或监听器接口.发布者可以动态注册某事件的订阅者,并在事件发生时通知

Before (homework)

```
class DivObserver {
  int m_div;
public:
  DivObserver( int div ) { m_div = div; }
  void update( int val ) {
    cout << val << " div " << m_div << " is "
         << val / m_div << '\n';
}_};

class ModObserver {
  int m_mod;
public:
  ModObserver( int mod ) { m_mod = mod; }
  void update( int val ) {
    cout << val << " mod " << m_mod << " is "
         << val % m_mod << '\n';
}_};
```

```
class Subject {
  int m_value;
  DivObserver m_div_obj;
  ModObserver m_mod_obj;
public:
  Subject() : m_div_obj(4), m_mod_obj(3) { }
  void set_value( int value ) {
    m_value = value;
    notify();
  }
  void notify() {
    m_div_obj.update( m_value );
    m_mod_obj.update( m_value );
}_};

int main( void ) {
  Subject subj;
  subj.set_value( 14 );
}

// 14 div 4 is 3
// 14 mod 3 is 2
```

After

- ??
- 将上述代码改造成Observer模式
- 给出代码和运行结果（截图）
- Deadline :下周2(12.29)