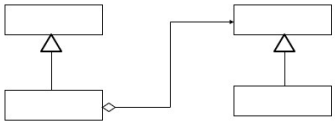
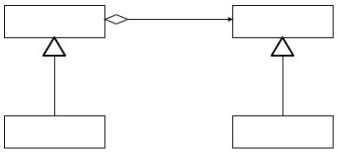
# 1 观察者模式 (Observer)

观察者模式是对象行为模式，又称为发布-订阅模式、模型-视图 (model-view) 模式、源-监听器模式或者从属者模式。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。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

根据观察者对象引用的存储地点，观察者模式的类图有微妙的区别。观察者模式类图（一）如下图所示:



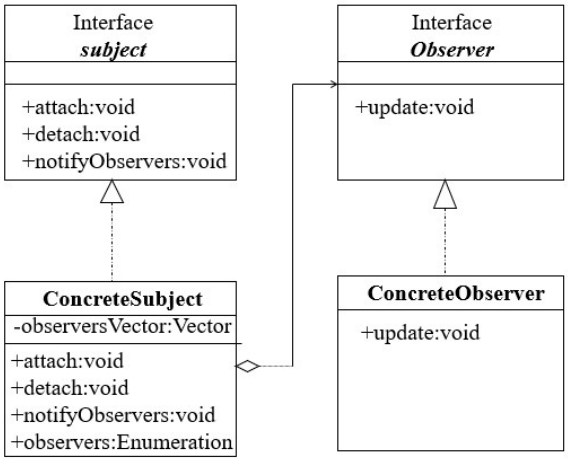
观察者模式的简略类图（二）如下图所示



Java 语言提供的观察者模式的实现属于此种结构.

## 1.1 观察者模式的结构

以一个简单的示意性实现为例，讨论观察者模式的结构。如下图:




可以看出，在这个观察者模式的实现里有下面这些角色:

- 抽象主题（Subject）角色：主题角色把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，主题角色又叫做抽象被观察者（Observable）角色。

- 具体主题（Concrete Subject）角色：负责管理具体主题的相关状态；在具体主题的状态改变时，给所有的观察者发出通知。具体主题又叫做具体被观察者 (Concrete Observable)。具体主题角色负责实现对观察者引用的聚集的管理方法。
- 抽象观察者（Observer）角色：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做通知/更新接口。
- 具体观察者（Concrete Observer) 角色：实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。

```
1  package com.javapatterns.observer;
2  public interface Subject {
3    //调用这个方法注册\登记一个新的观察者对象
4    public void attach(Observer observer);
5    //调用这个方法删除一个已经登记的观察者对象
6    public void detach(Observer observer);
7    //调用这个方法通知所有登记过的观察者对象
8    void notifyObservers();
9  }
```

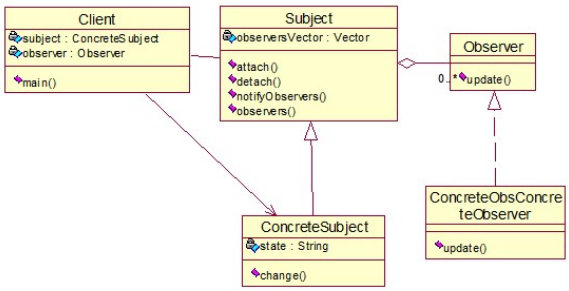
```
1  package com.javapatterns.observer;
2  import java.util.Vector;
3  import java.util.Enumeration;
4  public class ConcreteSubject implements Subject {
5    private Vector observersVector = new java.util.Vector();
6    public void attach(Observer observer) { //注册
7      observersVector.addElement(observer);
8    }
9    public void detach(Observer observer) {
10     observersVector.removeElement(observer);
11   }
12   public void notifyObservers() {
13     java.util.Enumeration enumeration = observers();
14     while (enumeration.hasMoreElements()) {
15       ((Observer)enumeration.nextElement()).update();
16     }
17   }
18   public Enumeration observers() {
19     return ((java.util.Vector) observersVector.clone()).elements();
20   }
21 }
```

```java
package com.javapatterns.observer;
public interface Observer {
   void update(); //调用这个方法会更新自己
}
```

```java
public class ConcreteObserver implements Observer {
  public void update() { //调用这个方法会更新自己
    System.out.println("I am notified.");
  }
}
```

### 1.1.1 另一种方案

- 考察主题对象的功能时, 发现它必需使用一个 java 聚集来维护一个所有的观察者对象的引用. 前一个方案中, 管理这个聚集的是具体主题, 因此在类图中存在从抽象观察者角色到具体主题角色的聚合连线.
- 但是所有具体主题管理聚集的方法都相同, 因此可以将它移到抽象主题中.
- 第二种实现方案和第一种实现方案的主要区别就是代表存储观察者对象的聚合连线是从抽象主题到抽象观察者.




```java
import java.util.Vector;
import java.util.Enumeration;
abstract public class Subject {
  //这个聚集保存了所有对观察者对象的引用
  private Vector observersVector = new java.util.Vector();
  //调用这个方法登记一个新的观察者对象
  public void attach(Observer observer) {
    observersVector.addElement(observer);
    System.out.println("Attached an observer.");
  }
  //调用这个方法删除一个已经登记过的观察者对象
```

```
  public void detach(Observer observer) {
    observersVector.removeElement(observer);
  }
  //调用这个方法通知所有登记过的观察者对象
  public void notifyObservers() {
    java.util.Enumeration enumeration = observers();
    while (enumeration.hasMoreElements()) {
      System.out.println("Before notifying");
    ((Observer)enumeration.nextElement()).update();
    }
  }
  //这个方法给出观察者聚集的Enumeration对象
  public Enumeration observers() {
    return ((java.util.Vector) observersVector.clone()).elements();
  }
}
```

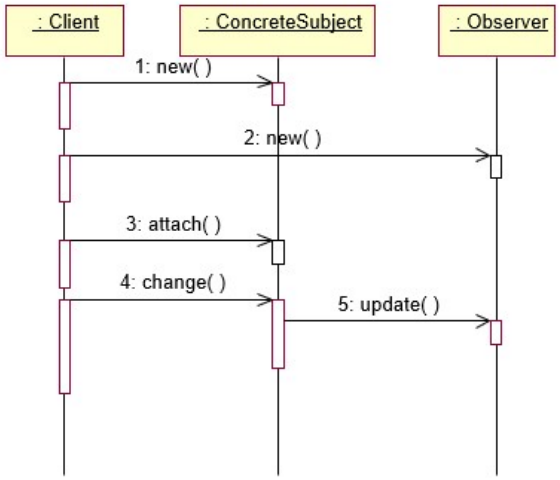
```
public class ConcreteSubject extends Subject {
  private String state;
  //调用这个方法更改主题的状态
  public void change(String newState) {
    state = newState;
    this.notifyObservers();
  }
}
```

抽象观察者角色的源代码没有变化, 不再列出.

```
package com.javapatterns.observer.variation;
public class Client {
  private static Subject subject;
  private static Observer observer;
  public static void main(String[] args) {
    subject = new ConcreteSubject();
    observer = new ConcreteObserver();
    subject.attach(observer);
    subject.change("new state");
  }
}
```

在运行时, 客户端首先创建了具体主题的实例, 以及一个观察者对象. 然后, 它调用主题对象的

attach() 方法, 将这个观察者对象向主题对象登记, 也就是将它加入到主题对象的聚集中去. 这时, 客户端调用主题的业务逻辑方法, 改变了主题对象的内部状态. 主题对象在状态发生变化时, 调用超类的 notifyObservers() 方法, 通知所有登记过的观察者对象. 其时序图如下:




## 1.2 Java 提供的对观察者模式的支持

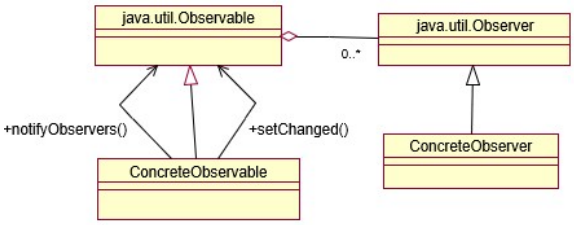
在 java 语言的 java.util 库里面, 提供了一个 Observable 类以及一个 Observer 接口, 构成 Java 语言对观察者模式的支持。

- Observer 接口: 这个接口只定义了一个方法, 即 update(Observable o, Object arg) 方法。当被观察者对象的状态发生变化时, 被观察者对象的 notifyObserver() 方法就会调用这一方法。
- Observable 类: 被观察者类都是 java.util.Observable 类的子类。java.util.Observable 提供公开的方法支持观察者对象, 这些方法中有两个对 Observable 子类非常重要:
  1. setChanged():setChanged() 被调用后会设置一个内部标记变量, 代表被观察者对象的状态发生了变化.
  2. notifyObservers(): 这个方法被调用时, 会调用所有登记过的观察者对象的 update() 方法, 使这些观察者可以更新自己.
  3. addObserver(): 将观察者对象加入到聚集中. 当有变化时, 这个聚集可以告诉 notifyObservers() 方法哪些观察者对象需要通知.

```
1 package java.util;
2 public class Observable {
3   private boolean changed = false;
4   private Vector obs;
5   //构造子,用零个观察者构造一个被观察者
6   public Observable() { obs = new Vector(); }
7   //将一个观察者加到观察者聚集上面
```

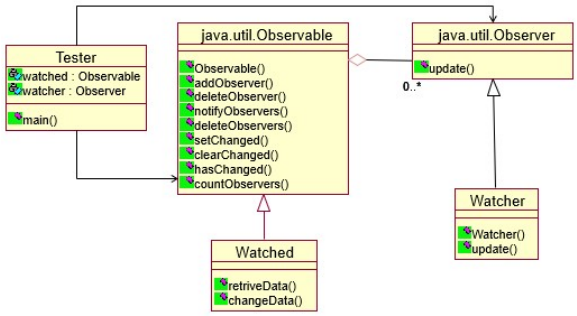
```
  public synchronized void addObserver(Observer o) {
    if (o == null) throw new NullPointerException();
    if (!obs.contains(o)) {
      obs.addElement(o);
    }
  }
  //将一个观察者对象从观察者聚集上删除
  public synchronized void deleteObserver(Observer o) {
    obs.removeElement(o);
  }
  //相当于notifyObservers(null)
  public void notifyObservers() { notifyObservers(null); }
  /**
   * 若本对象有变化(那时hasChanged方法会返回true)
   * 调用本方法通知所有登记的观察者,即调用它们的update()方法,
   * 传入参数this和arg作为参数
   */
  public void notifyObservers(Object arg) {
    //临时存放当前的观察者的状态,参见备忘录模式
    Object[] arrLocal;
    synchronized (this) {
      if (!changed) return;
      arrLocal = obs.toArray();
      clearChanged();
    }
    for (int i = arrLocal.length-1; i >= 0; i--)
      ((Observer)arrLocal[i]).update(this, arg);
  }
  //将观察者聚集清空
  public synchronized void deleteObservers() { obs.removeAllElements(); }
  //将“已变化”设为true
  protected synchronized void setChanged() { changed = true; }
  //将已变化重置为false
  protected synchronized void clearChanged() { changed = false; }
  //检测本对象是否已变化
  public synchronized boolean hasChanged() { return changed; }
  //返回被观察对象的观察者总数
  public synchronized int countObservers() { return obs.size(); }
}
```

Observable 也被称为主题对象. 一个被观察者对象有数个观察者对象, 每个观察者对象实现 Observer 接口. 被观察者对象发生变化时, 会调用 Observable 的 notifyObservers 方法, 此方法调用所有的具体观察者的 update() 方法, 从而使所有的观察者都被通知更新自己. 使用 java 语言提供的观察者模式的支持类图如下:




## 1.3 怎样使用 Java 对观察者模式的支持

本节给出一个非常简单的例子。在这个例子中，被观察者对象叫做 Watched；而观察者对象叫做 Watcher。Watched 对象继承自 java.util.Observable 类；而 Watcher 对象实现了 java.util.Observer 接口。另外有一个对象 Tester，扮演客户端的角色. 这个系统简单的结构图如下:




```
1  package com.javapatterns.observer.watching;
2  import java.util.Observable;
3  public class Watched extends Observable {
4    private String data = "";
5    public String retrieveData() { return data; }
6    public void changeData(String data) {
7      if (!this.data.equals(data)) {
8        this.data = data;
9        setChanged();
10     }
11     notifyObservers();
12   }
13 }
```

```
1  package com.javapatterns.observer.watching;
2  import java.util.Observable;
3  import java.util.Observer;
4  public class Watcher implements Observer {
5    public Watcher(Watched w) { w.addObserver(this); }
6    public void update( Observable ob, Object arg) {
7      System.out.println("Data has been changed to: '" +
8      ((Watched)ob).retrieveData() + "'");
9    }
10 }
```

```
1  package com.javapatterns.observer.watching;
2  import java.util.Observer;
3  public class Tester {
4    static private Watched watched;
5    static private Observer watcher;
6    public static void main(String[] args) {
7      watched = new Watched();
8      watcher = new Watcher(watched);
9      watched.changeData("In C, we create bugs.");
10     watched.changeData("In Java, we inherit bugs.");
11     watched.changeData("In Java, we inherit bugs.");
12     watched.changeData("In Visual Basic, we visualize bugs.");
13   }
14 }
```

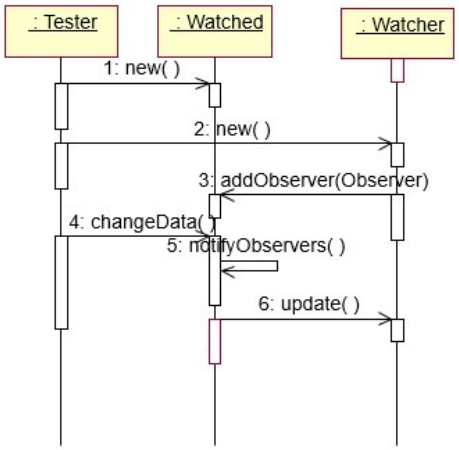
可以看出，虽然客户方将 Watched 对象的内部状态赋值了四次，但是值的改变只有三次，代码清单如下：

- watched.changeData(”In C, we create bugs.”);
- watched.changeData(”In Java, we inherit bugs.”);
- watched.changeData(”In Java, we inherit bugs.”);
- watched.changeData(”In Visual Basic, we visualize bugs.”);

相应地，Watcher 对象响应了三次改变。运行结果如下：

- Data has been changed to: ‘In C, we create bugs.’
- Data has been changed to: ‘In Java, we inherit bugs.’
- Data has been changed to: ‘In Visual Basic, we visualize bugs.’

系统的活动时序图如下：




Tester 对象首先创建了 Watched 和 Watcher 对象。在创建 Watcher 对象时，将 Watched 对象作为参量传入；然后 Tester 对象调用 Watched 对象的 changeData() 方法，触发 Watched 对象的内部状态变化；Watched 对象进而通知实现登记过的 Watcher 对象，也就是调用的 update() 方法。

## 1.4 Java 中的 DEM 事件机制

**AWT 中的 DEM 机制**: 在 AWT1.1 版本以及以后的各个版本中，事件处理模型均为基于观察者模式的委派事件模型（Delegation Event Model 或 DEM）。在 DEM 中，主题角色负责发布事件，观察者角色向特定的主题订阅它所感兴趣的事件。当一个具体主题产生一个事件时，它就会通知所有感兴趣的订阅者。在 DEM 模型中发布者叫做事件源，而订阅者叫做事件监听器。

```
new java.awt.Button(new ActionListner(){
  void actionPerform(){
    // ...
  }
});
```

## 1.5 观察者模式的优缺点

- 优点：观察者模式在被观察者和观察者之间建立一个抽象的耦合。观察者模式支持广播通信。
- 缺点：如果一个被观察者有很多的观察者，将所有的观察者都通知到会花费很多时间。如果在被观察者之间有循环依赖，被观察者会触发它们之间进行循环调用，导致系统崩溃。虽然观察者随时知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎样发生变化的。