广东技术师范大学
Guangdong Polytechnic Normal University

《系统分析与设计》课程

Instructor: Wen Jianfeng
Email: wjfgdin@qq.com

系统分析与设计

第7讲：设计模式--行为型

提纲

- 策略模式（Strategy）
- 观察者模式（Observer）

		Purpose 目的		
		Creational 创建型（**5**种）	**Structural** 结构型（**7**种）	**Behavioral** 行为型（**11**种）
Scope 范围	**Class** 类	Factory Method [工厂方法]	Adapter (class) [适配器（类）]	Interpreter [解释器] Template Method [模板方法]
	Object 对象	Abstract Factory [抽象工厂] Builder [建造者 / 生成器] Prototype [原型] Singleton [单例]	Adapter (object) [适配器（对象）] Bridge [桥接] Composite [组合] Decorator [装饰器] Façade [门面 / 外观] Flyweight [享元] Proxy [代理]	Chain of Responsibility [责任链 / 职责链] Command [命令] Iterator [迭代器] Mediator [中介者] Memento [备忘录] **Observer** [观察者] State [状态] **Strategy** [策略] Visitor [访问者]

设计模式-行为型

- It is concerned with assignment of responsibilities between the objects. What makes them different from structural patterns is they don't just specify the structure but also outline the patterns for message passing/communication between them. Or in other words, they assist in answering "How to run a behavior in software component?“

  行为型设计模式关注对象的职责分配。

  与结构型设计模式的不同之处在于它们不仅指定了对象结构，还指明了它们之间的消息传递/通信模式。换句话说，它们协助回答“在软件组件中如何执行一个行为？”的问题。

策略模式
Strategy Pattern

策略模式-现实世界中的例子

- Consider the example of sorting, we implemented bubble sort but the data started to grow and bubble sort started getting very slow. In order to tackle this we implemented Quick sort. But now although the quick sort algorithm was doing better for large datasets, it was very slow for smaller datasets. In order to handle this we implemented a strategy where for small datasets, bubble sort will be used and for larger, quick sort.

  例如：冒泡排序算法在数据量小时速度很快，但数据量增大时变得非常缓慢。快速排序算法则对大型数据集效果更好。因此，对于小型数据集，我们选用冒泡排序，大型数据集则选用快速排序。

策略模式 – Intent【意图】

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

  定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换

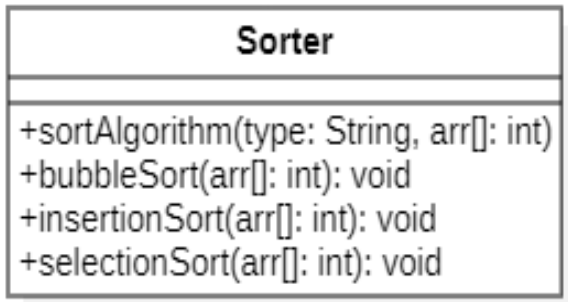
  Strategy模式使得算法可独立于使用它的客户而变化

策略模式 – Motivation【动机】

- 在软件系统中，有许多算法可以实现某一功能（如查找/排序的算法），一种传统的做法是将算法硬编码在一个类中
  - 如：需要提供多种算法，可以将这些算法写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的算法；当然也可以将这些算法封装在一个统一的方法中，通过if…else…等条件判断语句来进行选择

策略模式 – Motivation【动机】

- 例如：对一个数组中的整数进行排序
  - 传统的设计方案




```
public class Sorter {
  public static int[] sortAlgorithm(String type, int[] arr) {
      if (type == "bubble") { // 冒泡排序算法
    bubbleSort(arr);
      } else if (type == "insertion") { // 插入排序算法
          insertionSort(arr);
      } else if (type == "selection") { // 选择排序算法
          selectionSort(arr);
      }
      return arr;
  } ......
```

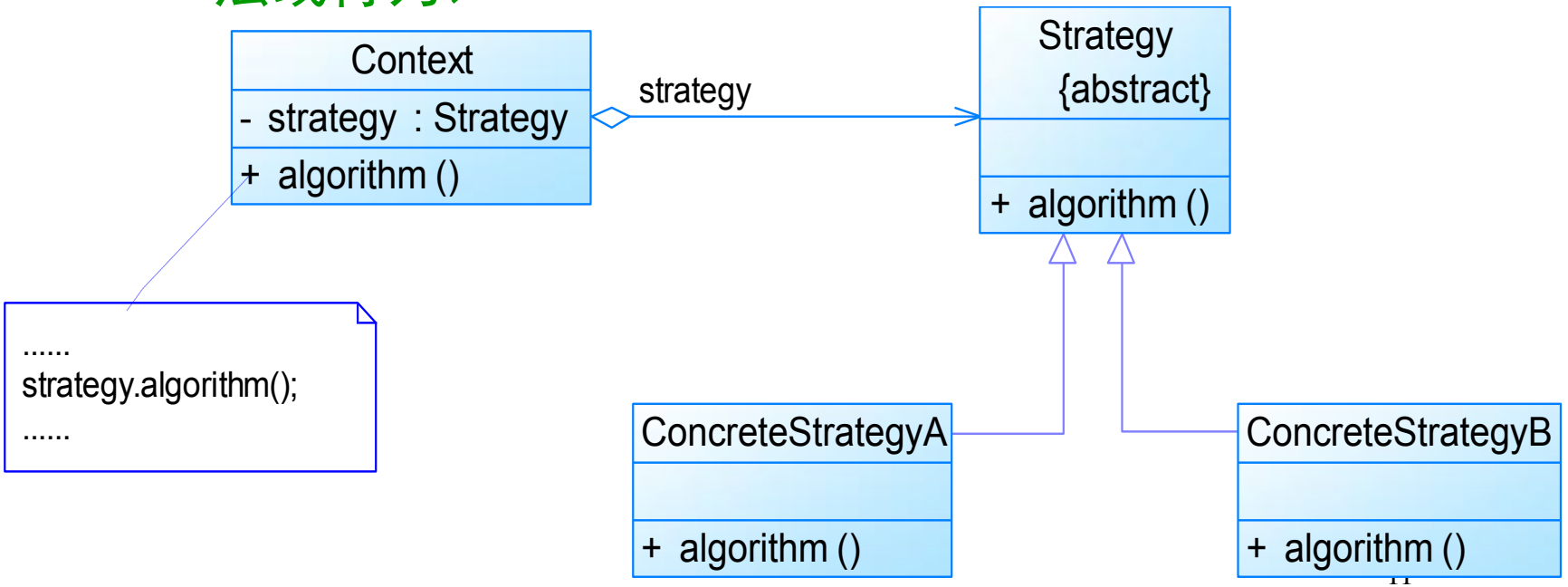
  - 传统方法的缺点
    - 所有的算法都在一个类中实现，导致一个类的代码太多，影响可读性、可维护性
    - 当增加一种新的算法时，需要修改封装算法类的源代码
    - 当更换算法时，需要修改客户端调用代码

策略模式 – Motivation【动机】

- 解决方法：使用策略模式
  - 可以定义一些独立的类来封装不同的算法，每一个类封装一个具体的算法，在这里，每一个封装算法的类我们都可以称之为策略（Strategy）。为了保证这些策略的一致性，一般会用一个抽象的策略类来做算法的定义，而具体每种算法则对应于一个具体策略类

策略模式 – Structure【结构】

- 策略模式涉及3个角色
  - Context—环境角色（持有一个Strategy类的引用）
  - Strategy—抽象策略角色（给出所有具体策略类所需的接口）
  - Concrete Strategy—具体策略角色（包装了相关的算法或行为）

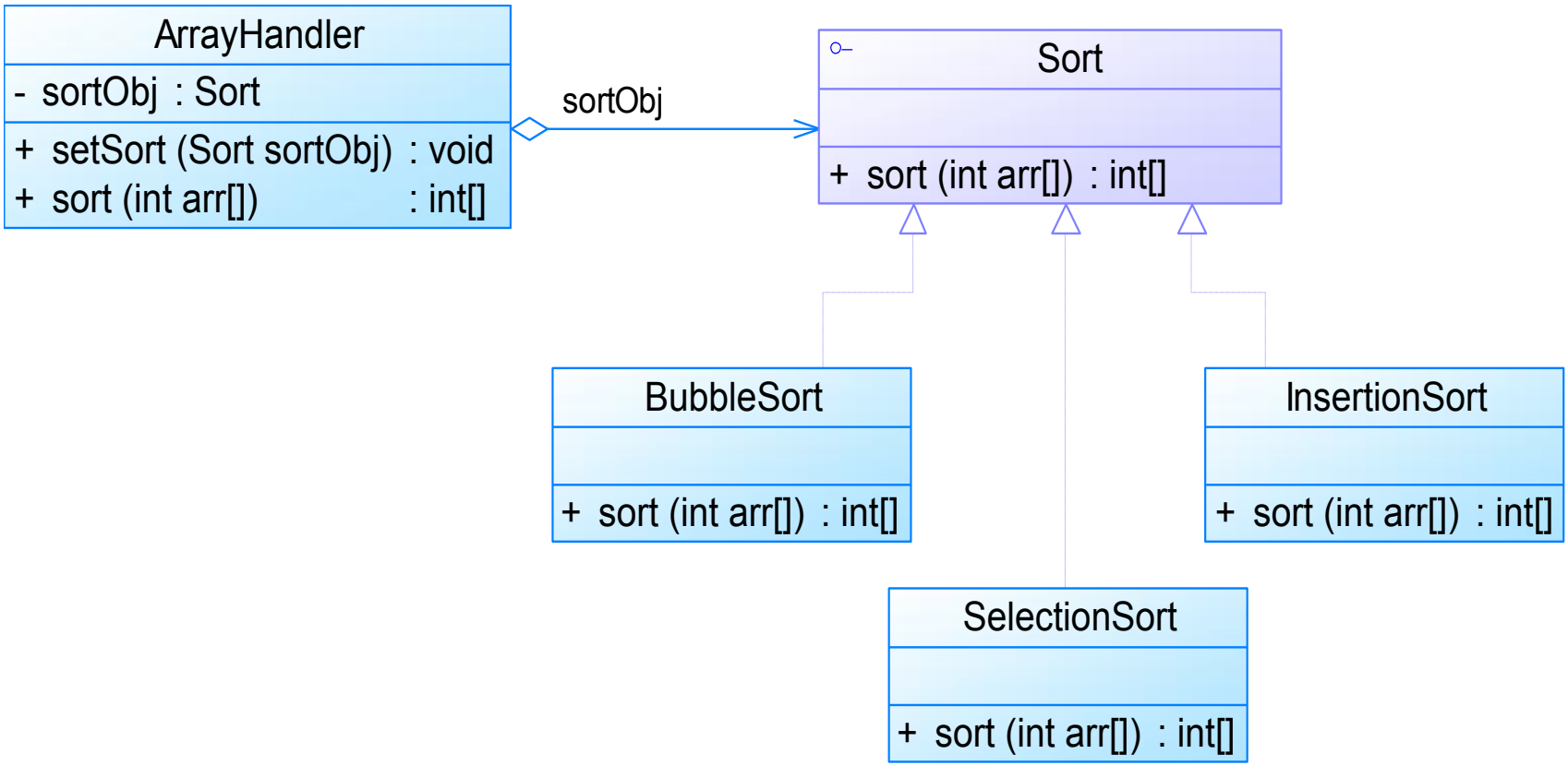


策略模式 – 【举例】

- 举例：排序策略
  - 实现对一个数组中的整数进行排序，使得客户可以动态地更换排序算法，如可选择冒泡排序、选择排序、插入排序、快速排序等算法，同时能够灵活地增加新的排序算法

策略模式 – 【举例】

- 使用策略模式进行设计



策略模式 – 【举例】

```
/**
 * 排序策略接口
 */
public interface Sort {
   /**
    * 抽象的排序方法
    */
   int[] sort(int arr[]);
}
```

```
/* 实现冒泡排序算法的类 */
public class BubbleSort implements Sort {
  /** sort()的具体实现之一：冒泡排序 */
  public int[] sort(int arr[]) {
    int len = arr.length;
    for (int i = 0; i < len; i++) {
      for (int j = i + 1; j < len; j++) {
        int temp;
        if (arr[i] > arr[j]) {
          temp = arr[j];
          arr[j] = arr[i];
          arr[i] = temp;
        }
      }
    }
    System.out.println("冒泡排序");
    return arr;
  }
}
```

策略模式 – 【举例】

```
/* 环境（上下文）类 */
public class ArrayHandler {
   private Sort sortObj; // 排序算法对象

   /* 使用所设置的排序算法进行排序 */
   public int[] sort(int arr[]) {
        sortObj.sort(arr);
        return arr;
   }

   /* 设置具体使用的排序算法 */
   public void setSortObj(Sort sortObj) {
        this.sortObj = sortObj;
   }
}
```

策略模式 – 【举例】

```
public class Client { // 客户程序
  public static void main(String args[]) {
    int arr[] = { 1, 4, 6, 2, 5, 3, 7, 10, 9 };
    int result[];

    ArrayHandler ah = new ArrayHandler(); // 环境（上下文）对象

    Sort sort; // 排序算法对象
    sort = new SelectionSort(); // 客户决定使用选择排序算法

    ah.setSortObj(sort); // 在ArrayHandler类中设置具体排序算法
    result = ah.sort(arr); // 执行排序

    for (int i = 0; i < result.length; i++) {
      System.out.print(result[i] + ",");
    }
  }
}
```

策略模式 – 【举例】

- 也可进一步优化
  - 每种排序类都是无状态的，没必要在每次使用的时候，都重新创建一个新的对象。可以使用工厂模式对对象的创建进行封装

```
public class SortAlgorithmFactory {
  private static final Map<String, Sort> algs = new HashMap<>();
  static {
    algs.put("BubbleSort", new BubbleSort());
    algs.put("InsertionSort", new InsertionSort());
    algs.put("SelectionSort", new SelectionSort());
    algs.put("QuickSort", new QuickSort());
  }
  public static Sort getSortAlgorithm(String type) {
    if (type == null || type.isEmpty()) {
      throw new IllegalArgumentException("type should not be empty.");
    }
    return algs.get(type);
  }}
```

客户代码的修改：

```
sort = SortAlgorithmFactory.getSortAlgorithm("InsertionSort");
```

策略模式 – 【优缺点】

- 优点：
  - 符合“开闭原则”，客户可灵活选择算法或行为，也可灵活增加新的算法或行为，而不影响已有的代码
  - 提供了管理相关算法族的办法，易于重用各算法的共同功能
  - 提供了可以替代继承关系的办法，避免直接继承Context类导致的复杂性
  - 消除了多重条件语句
  - 提供对相同行为的不同具体实现，客户可自由选择不同的实现
- 缺点：
  - 客户必须理解所有的策略类才能作出合适的选择
  - 产生很多策略类，对象数目增加

策略模式 – 【适用情形】

- 适用情形
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

    许多相关的类仅仅是行为不同。策略模式为每个行为配置一个类
  - You need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

    需要使用一个算法的不同变种。当要将这些变种实现为一个算法的类层次结构时，可以使用策略模式

策略模式 – 【适用情形】

- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

  算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构

- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

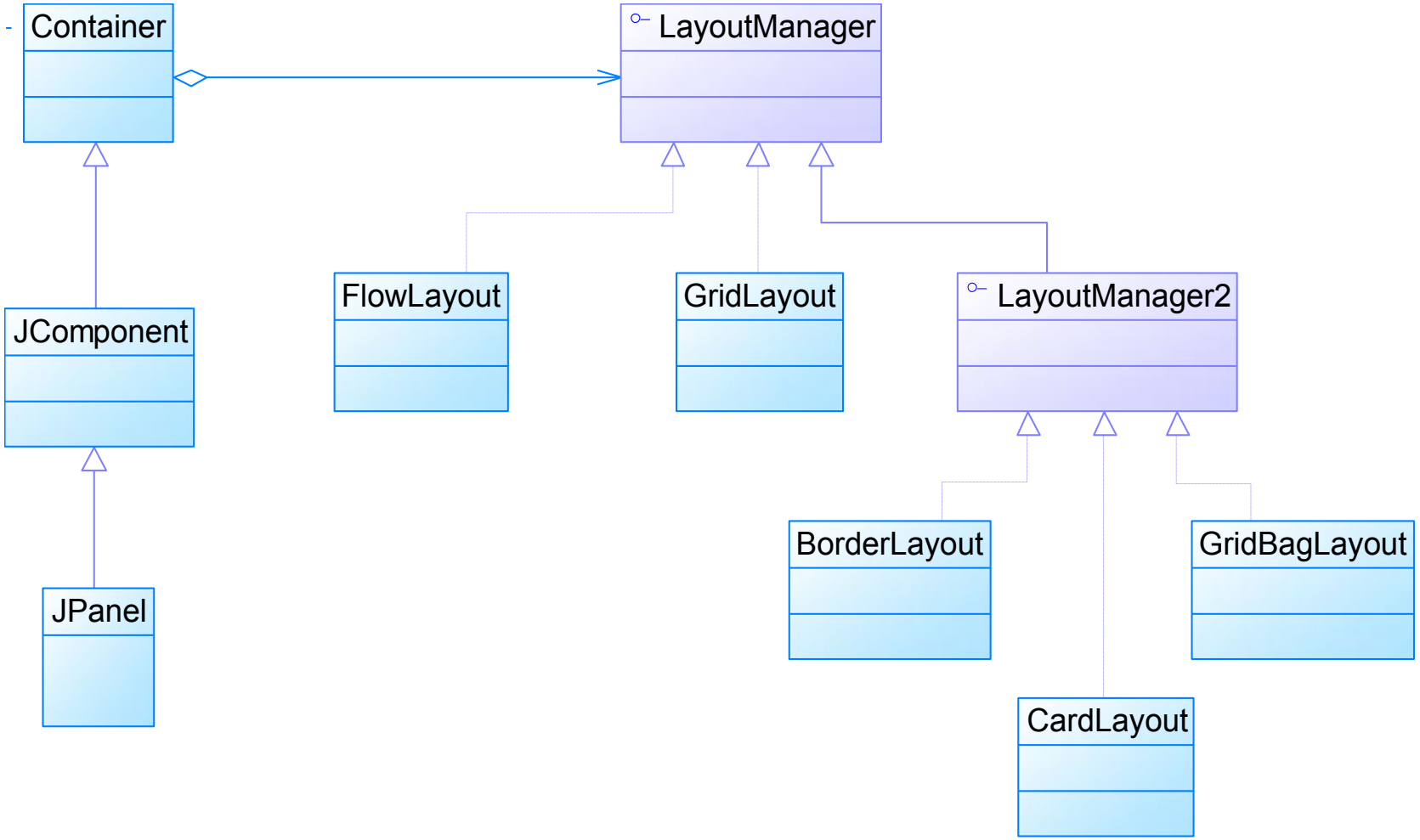
  一个类定义了多种行为, 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句

策略模式 – 在JDK中的应用

- Java JDK中的策略模式
  - 布局管理器LayoutManager（见下页的类图）
    - Context角色：Container类
    - Strategy角色：LayoutManager接口
    - Concrete Strategy角色：BorderLayout, FlowLayout, … …
  - Swing中的组件边框Border
    - Context角色：JComponent类
    - Strategy角色：Border接口
    - Concrete Strategy角色：TitledBorder, LineBorder, … …

策略模式 – 在JDK中的应用

- JDK中的布局管理器LayoutManager



观察者模式
Observer Pattern

观察者模式 – Intent【意图】

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

- 观察者模式也叫：
  - 发布-订阅模式（Publish-Subscribe）
  - 模型-视图模式（Model-View）
  - 源-监听器模式（Source-Listener）
  - 从属者模式（Dependents）

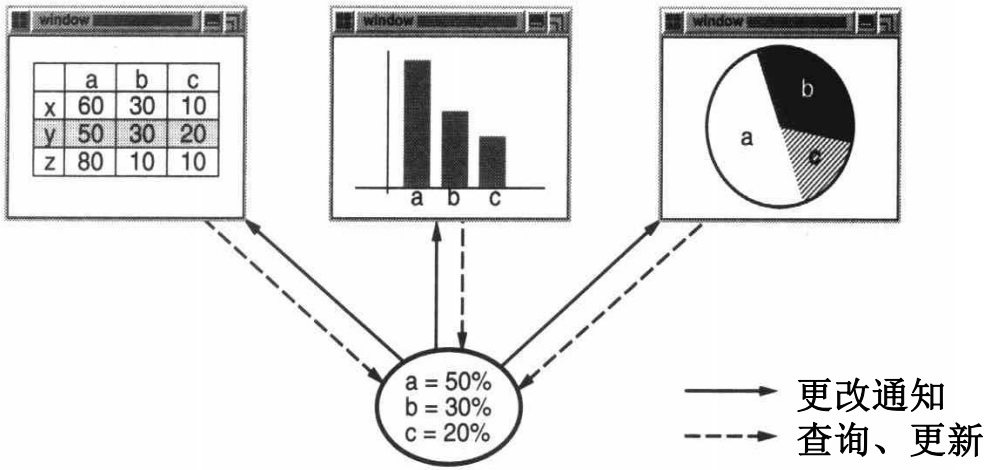
观察者模式-现实世界的例子

- A good example would be the job seekers where they subscribe to some job posting site and they are notified whenever there is a matching job opportunity.

  一个很好的例子是求职者，他们订阅了一些职位发布网站，只要有匹配的工作机会，他们就会得到通知。

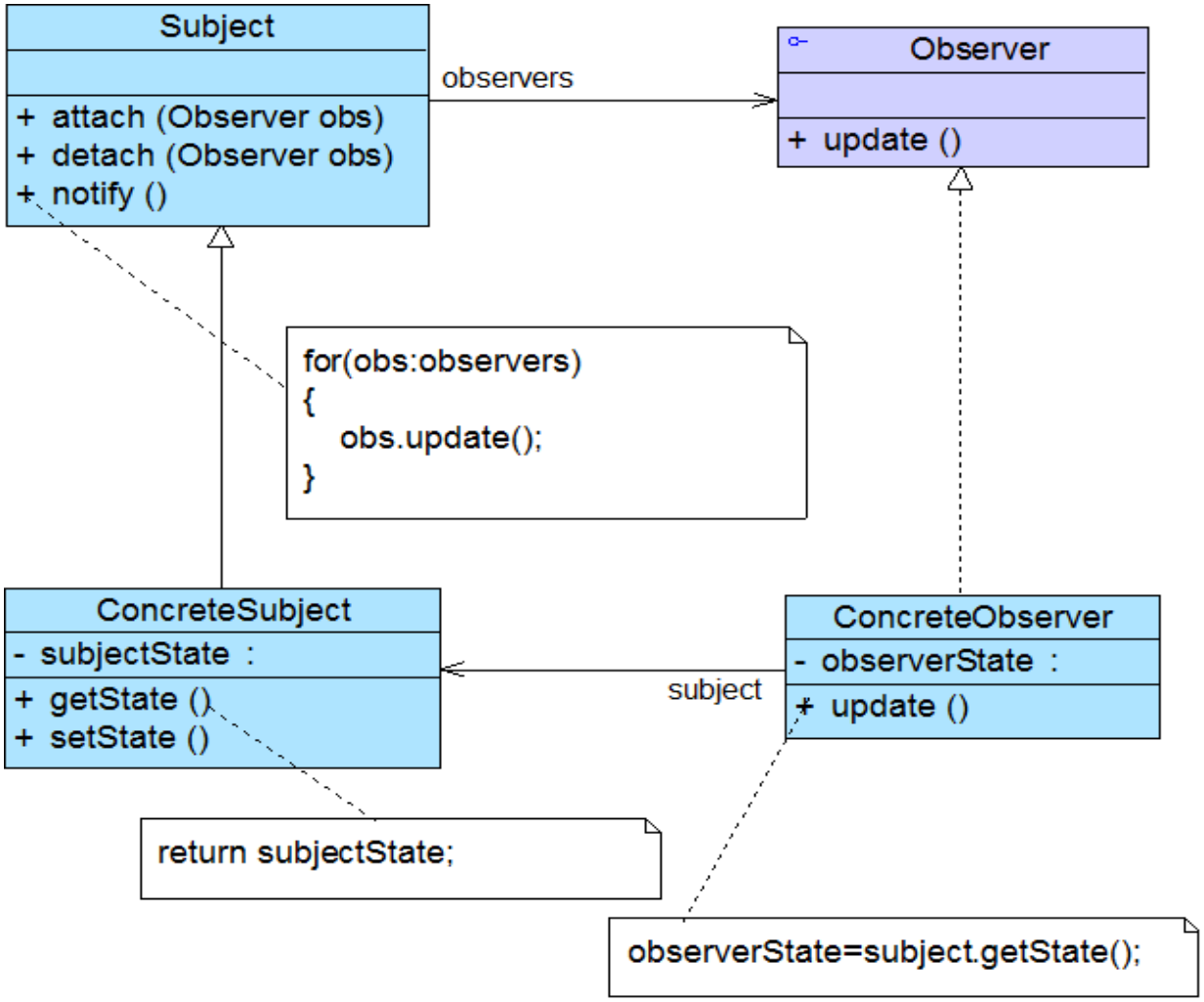
观察者模式 – Motivation【动机】

- 建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应
  - 发生改变的对象称为观察目标（主题），而被通知的对象称为观察者，一个观察目标（主题）可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展



观察者模式 – Structure【结构】

- 观察者模式的设计类图

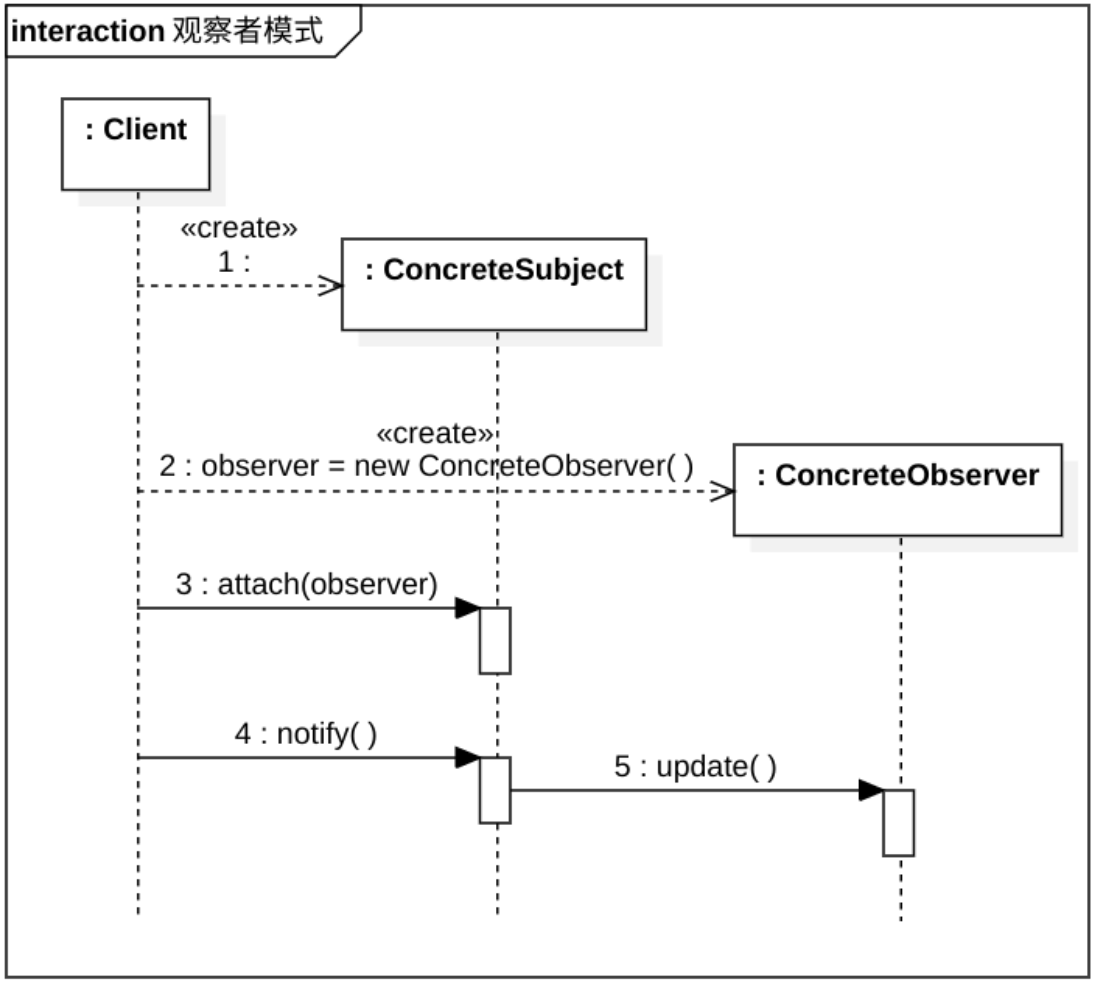


观察者模式 – Structure【结构】

- 观察者模式涉及的4个角色
  - Subject—抽象主题角色
    - 知悉它的观察者，可以拥有任意数量的观察者
    - 提供增加和删除观察者对象的接口
  - Observer—抽象观察者角色
    - 为观察者定义一个 updating 接口，当主题发生变化时通知观察者更新
  - ConcreteSubject—具体主题角色
    - 将有关状态存入具体观察者对象
    - 在具体主题的内部状态改变时，给所有登记过的观察者发出通知
  - ConcreteObserver—具体观察者角色
    - 持有到具体主题对象的引用
    - 存储与主题一致的状态
    - 实现 updating 接口以保持观察者的状态与主题一致

观察者模式 – Structure【结构】

- 观察者模式的实现顺序图



观察者模式 – 【举例一】

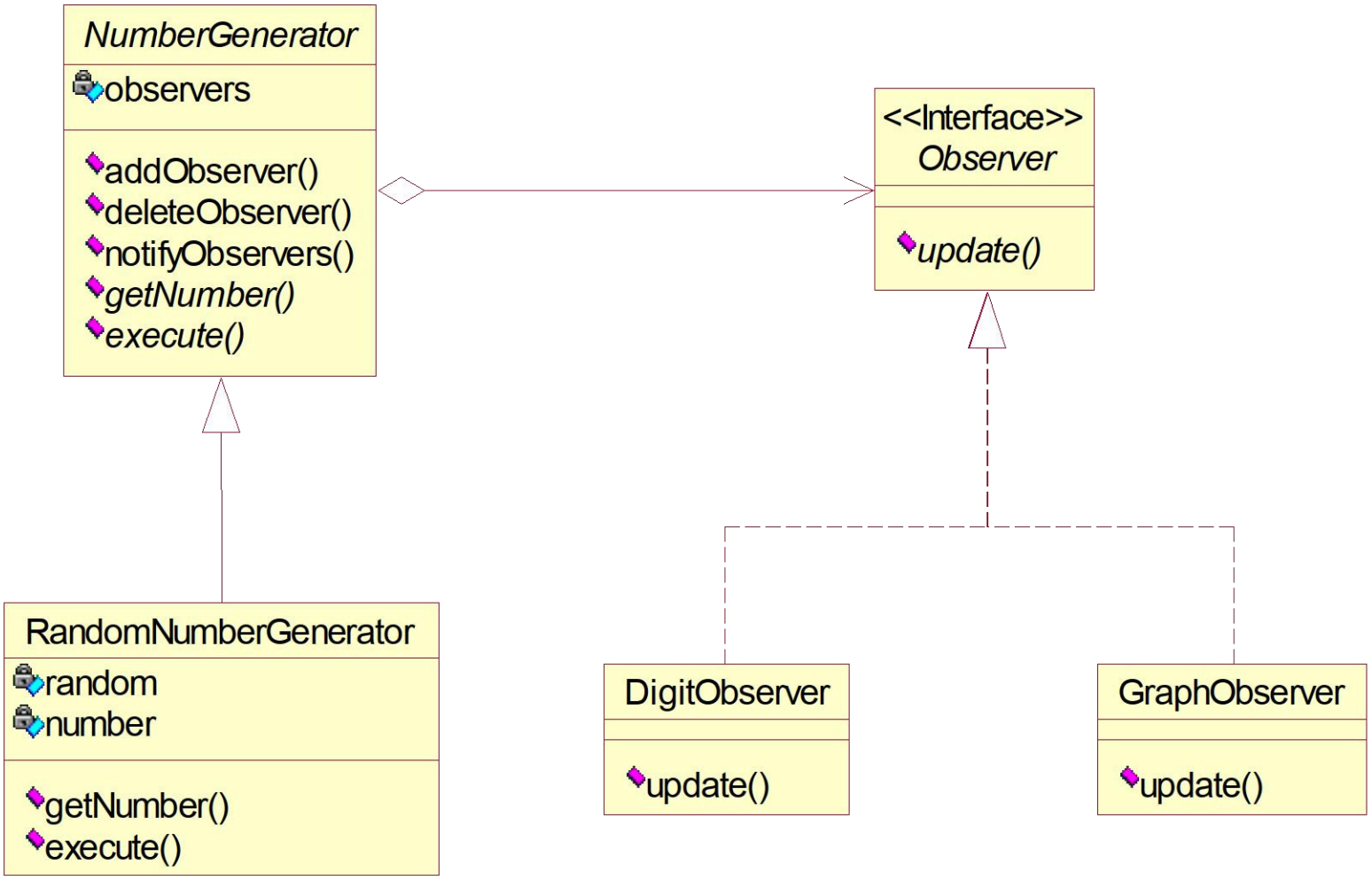
- 举例：观察产生多个数值的对象，然后输出该值。输出方式因观察者而异：
  - DigitObserver以数字表示数值
  - GraphObserver以简易长条图表示数值
  - 当观察的目标值发生变化时，自动通知观察者

  - 运行效果如下：

```
DigitObserver: 5
GraphObserver: *****

DigitObserver: 2
GraphObserver: **
```

观察者模式 – 【举例一】

- 设计类图



观察者模式 – 【举例一】

- Subject

```
/* 表示产生数值对象的抽象类，即抽象主题类Subject */
public abstract class NumberGenerator {
   /** 在数组中存储Observers对象 */
   private ArrayList<Observer> observers = new
   ArrayList<Observer>();

   /** 新增Observer */
   public void addObserver(Observer observer) {
        observers.add(observer);
   }
   /** 通知所有Observers更新数据 */
   public void notifyObservers() {
        for (Observer obs : observers)
             obs.update(this);
   }

   public abstract int getNumber(); // 取得数值
   public abstract void execute(); // 产生数值
}
```

观察者模式 – 【举例一】

- ConcreteSubject

```
/** 产生随机数的类，即具体主题类ConcreteSubject */
public class RandomNumberGenerator extends NumberGenerator {
   private Random random = new Random(); // 随机数产生器
   private int number; // 当前数值

   /** 取得数值 */
   public int getNumber() {
        return number;
   }
   /** 产生随机数，并通知观察者数据发生了变化 */
   public void execute() {
        for (int i = 0; i < 5; i++) {
                number = random.nextInt(20);
                notifyObservers(); // 该方法在父类中已实现
        }
   }
}
```

观察者模式 – 【举例一】

- Observer & ConcreteObservers

```
/** 表示观察者的接口 */
public interface Observer {
   void update(NumberGenerator generator);
}
```

```
/** 以数字表示数值的类，即具体观察者类ConcreteObserver */
public class DigitObserver implements Observer {
  public void update(NumberGenerator generator) {
    System.out.println("DigitObserver:" + generator.getNumber());
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

观察者模式 – 【举例一】

- Observer & ConcreteObservers

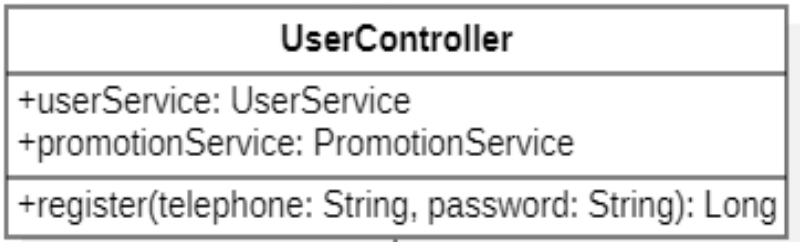
```
/** 以简易长条图表示数值的类，即具体观察者类ConcreteObserver */
public class GraphObserver implements Observer {
   public void update(NumberGenerator generator) {
        System.out.print("GraphObserver:");
        int count = generator.getNumber();
        for (int i = 0; i < count; i++) {
                System.out.print(“*”); // 打印星号
        }
        System.out.println();

        try {
                Thread.sleep(500);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
   }
}
```

观察者模式 – 【举例一】

```
// 测试用的类
public class Test {
  public static void main(String[] args) {
    // 创建具体主题对象
    NumberGenerator generator = new RandomNumberGenerator();

    // 创建观察者对象
    Observer observer1 = new DigitObserver();
    Observer observer2 = new GraphObserver();

    // 注册观察者
    generator.addObserver(observer1);
    generator.addObserver(observer2);

    // 产生数据，并通知观察者
    generator.execute();
  }
}
```

观察者模式 – 【举例二】

- 举例二：
  - 假设开发一个投资理财系统，用户注册成功之后，给他发放投资体验金



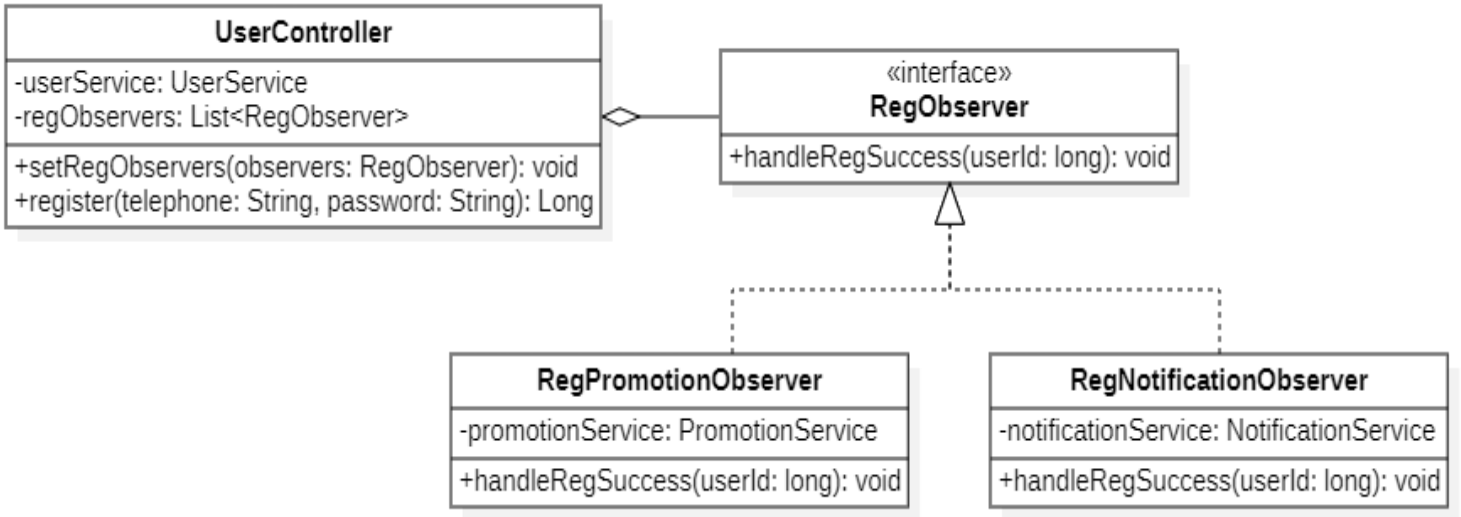

```
public class UserController {
  private UserService userService;
  private PromotionService promotionService;

  public Long register(String telephone, String password) {
    // 注册
    long userId = userService.register(telephone, password);
    // 发放投资体验金
    promotionService.issueNewUserExperienceCash(userId);
    return userId;
  }
}
```

观察者模式 – 【举例二】

- 未使用观察者模式的初始设计：
  - 该类做了两件事情，注册和发放体验金
  - 虽违反单一职责原则，但如果无扩展需求也是可接受的
  - 相反，如果需求频繁变动，比如，用户注册成功之后，不再发放体验金，而是改为发放优惠券，并且还要给用户发送一封“注册成功”的站内信。
  - 这种情况下，我们就需要频繁地修改 register() 函数中的代码，违反开闭原则

观察者模式 – 【举例二】

- 使用观察者模式的设计



- 当需要添加新的观察者的时候，UserController 类的 register() 函数完全不需要修改，只需要再添加一个实现了 RegObserver 接口的类，并且通过 setRegObservers() 函数将它添加到 UserController 类中即可，符合开闭原则

```
public interface RegObserver {
  void handleRegSuccess(long userId);
}
```

```
public class RegPromotionObserver implements RegObserver {
  private PromotionService promotionService =
                                        new PromotionService();
  @Override
  public void handleRegSuccess(long userId) {
    promotionService.issueNewUserExperienceCash(userId);
  }
}
```

```
public class RegNotificationObserver implements RegObserver {
  private NotificationService notificationService =
                                      new NotificationService();
  @Override
  public void handleRegSuccess(long userId) {
    notificationService.sendInboxMessage(userId, "欢迎您");
  }
}
```

```
public class UserController {
  private UserService userService = new UserService();
  private List<RegObserver> regObservers = new ArrayList<>();

  public void setRegObservers(RegObserver observers) {
    regObservers.add(observers);
  }

  public Long register(String telephone, String password) {
    long userId = userService.register(telephone, password);

    for (RegObserver observer : regObservers) {
      observer.handleRegSuccess(userId);
    }

    return userId;
  }
}
```

```
public class Demo {
  public static void main(String[] args) {
    // 创建主题对象
    UserController userController = new UserController();

    // 创建观察者对象
    RegObserver observer1 = new RegPromotionObserver();
    RegObserver observer2 = new RegNotificationObserver();

    // 在主题对象中添加观察者对象
    userController.setRegObservers(observer1);
    userController.setRegObservers(observer2);

    // 用户注册
    userController.register("13512345678", "password");
  }
}
```

观察者模式 – 【举例二】

- 可进一步优化
  - 问题：前面的实现方式，是一种同步阻塞的实现方式，可能存在性能问题
    - 观察者和被观察者代码在同一个线程内执行，被观察者一直阻塞，直到所有的观察者代码都执行完成之后，才执行后续的代码
    - 即：register() 函数依次调用执行每个观察者的 handleRegSuccess() 函数，等到都执行完成之后，才会返回结果给客户端
  - 优化思路：如果 register() 是一个调用比较频繁的接口，对性能非常敏感，可以将同步阻塞的实现方式改为异步非阻塞的实现方式，以此来减少响应时间
    - 即：当 userService.register() 函数执行完成之后，启动一个新的线程来执行观察者的 handleRegSuccess() 函数

观察者模式 – 【举例二】

- 优化方式一
  - 在每个 handleRegSuccess() 函数中创建一个新的线程

```
public class RegPromotionObserver implements RegObserver {
  ... ...
  // 在新线程中执行
  @Override
  public void handleRegSuccess(long userId) {
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        promotionService.issueNewUserExperienceCash(userId);
      }
    });
  thread.start();
  }
}
```

观察者模式 – 【举例二】

- 优化方式二
  - 在 UserController 的 register() 函数中使用线程池来执行每个观察者的 handleRegSuccess() 函数
  - 代码见下页... ...

```
public class UserController {
  ... ...
  private Executor executor;

  public UserController(Executor executor) {
    this.executor = executor;
  }

  ... ...

  // 使用线程池的方式
  public Long register(String telephone, String password) {
    long userId = userService.register(telephone, password);
    for (RegObserver observer : regObservers) {
      executor.execute(new Runnable() {
        @Override
        public void run() {
          observer.handleRegSuccess(userId);
        }
      });
    }
    return userId;
  }
}
```

观察者模式 – 【优缺点】

- 优点
  - 观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色
  - Subjects 和 Observers 可以独立变化、可以独立重用，添加 Observer 不影响 Subjects 和其他 Observers，符合“开闭原则”
  - Subject 和 Observer 之间的耦合是抽象耦合
  - 支持广播通信，由 Observer 决定处理还是忽略 Subject 的通知

观察者模式 – 【优缺点】

- 缺点
  - 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间
  - 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃
  - 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化

观察者模式 – 【适用情形】

- 适用情形
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

    当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用

  - When a change to one object requires changing others, and you don't know how many objects need to be changed.

    当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变

观察者模式 – 【适用情形】

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

  当一个对象必须通知其它对象，而它又不能假定其它对象是谁。
  换言之，你不希望这些对象是紧密耦合的

观察者模式 – 在JDK中的应用

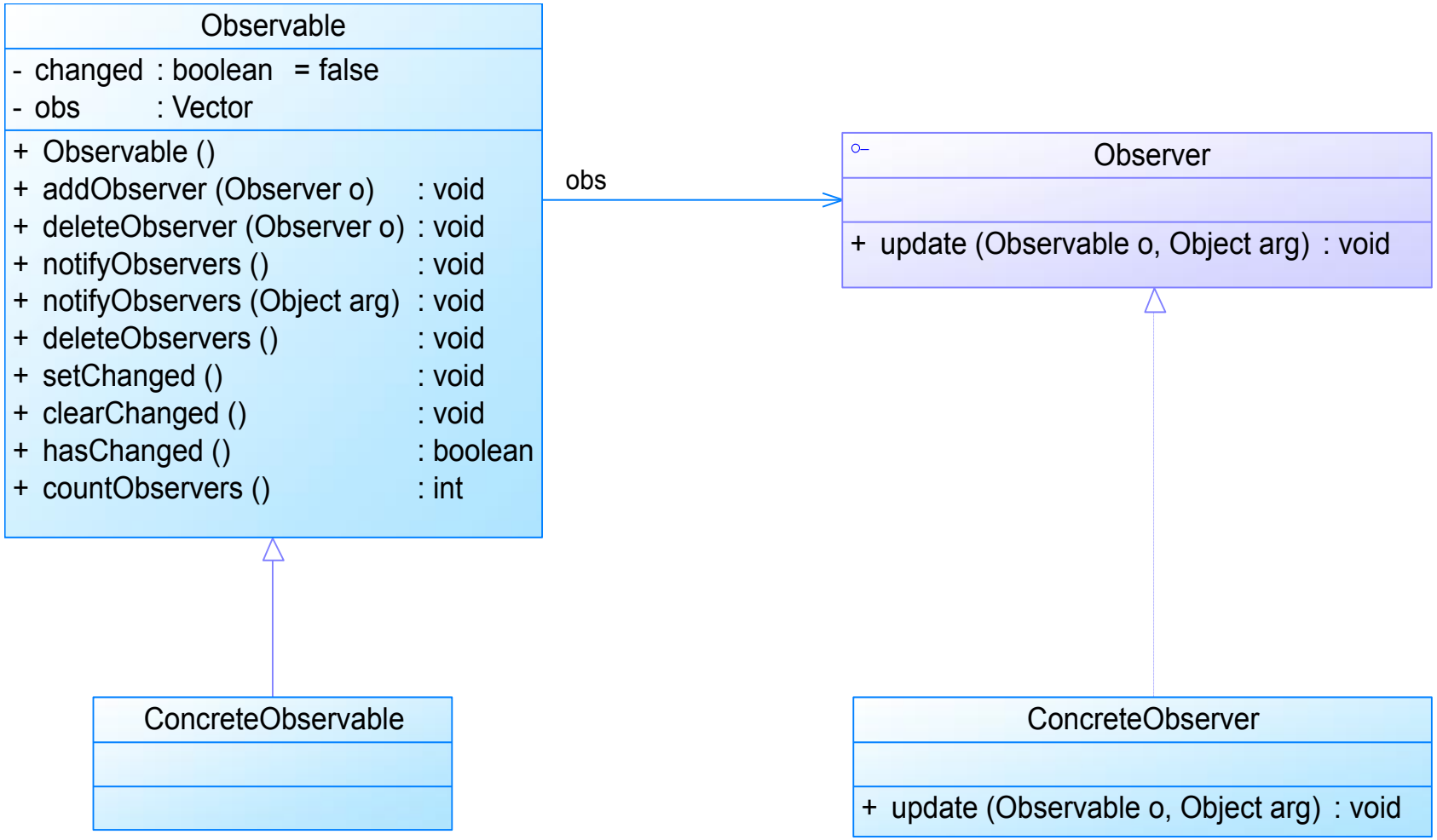
- Java JDK中的例子
  - GUI的事件处理机制
    - 事件源保持一个事件监听器的列表
- JDK本身也提供了对观察者模式的支持
  - 注：JDK9后标记为弃用，官方建议使用java.beans中的事件模型和java.util.concurrent的并发数据结构
  - java.util.Observer

void	update(Observable o, Object arg) 只要改变了 observable 对象就调用此方法。

  - java.util.Observable

void	addObserver(Observer o) 如果观察者与集合中已有的观察者不同，则向对象的观察者集中添加此观察者。

观察者模式 – 在JDK中的应用



Thank you!