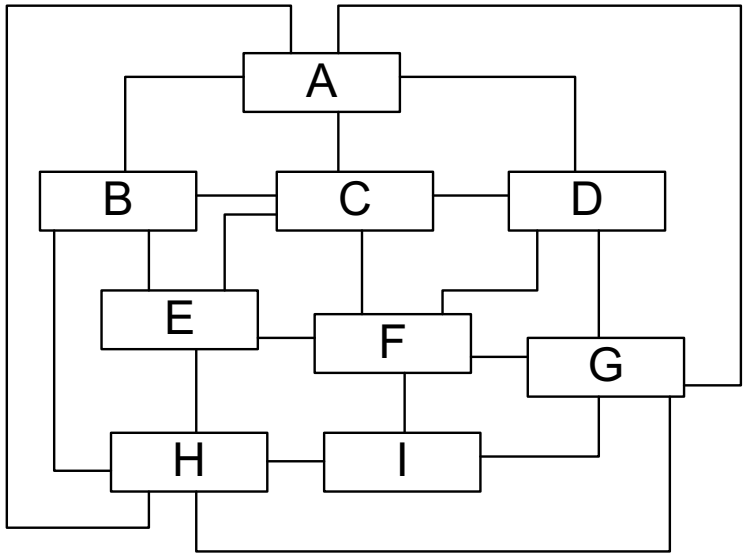
南京大学

设计模式–行为型模式(二)

Design Pattern-Behavioral Pattern (2)

中介者模式概述

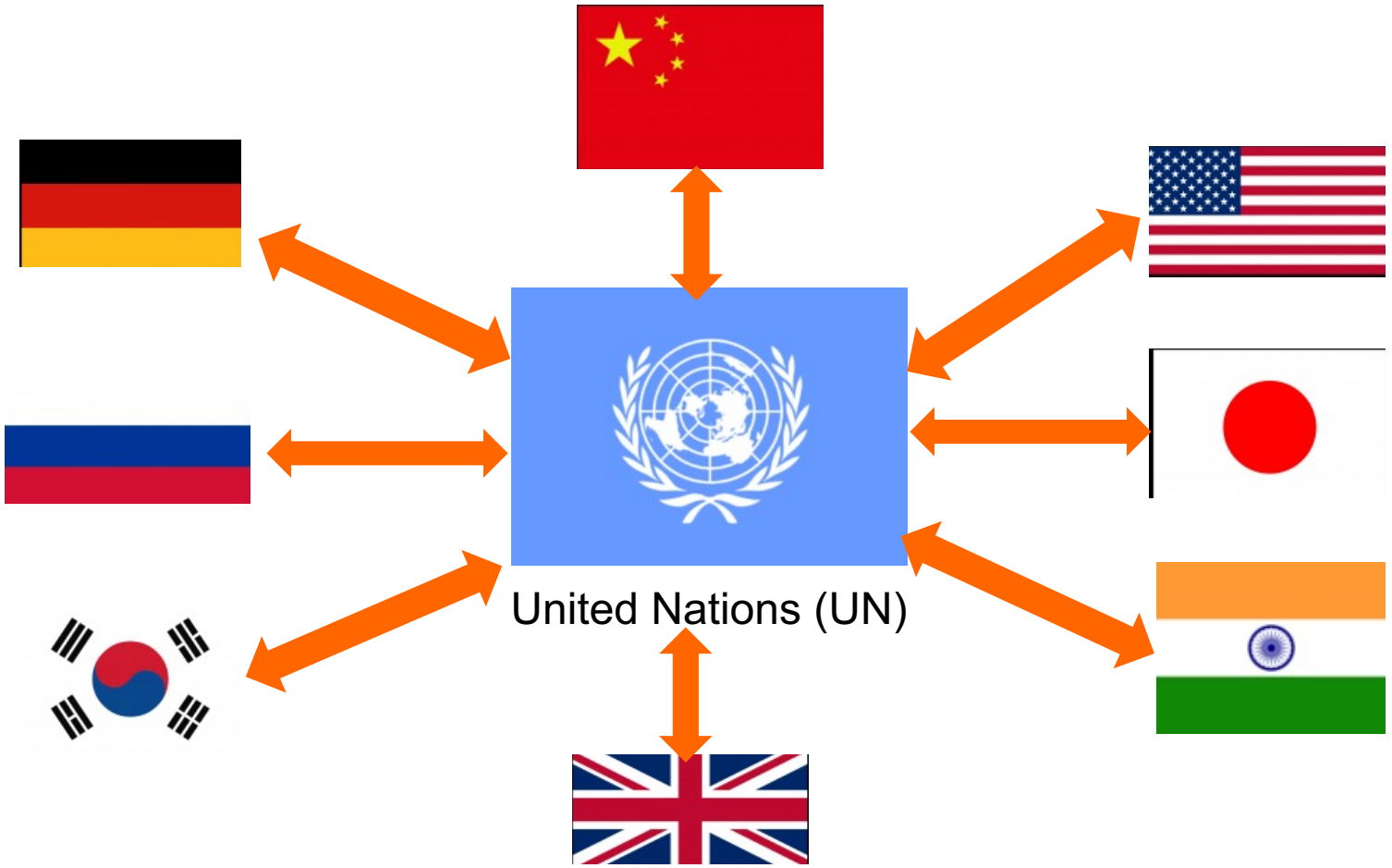
- 分析
  - 软件开发：
    - **网状结构**：多对多联系将导致系统非常复杂，几乎每个对象都需要与其他对象发生相互作用，而这种相互作用表现为一个对象与另外一个对象的直接耦合，这将导致一个过度耦合的系统



网状结构

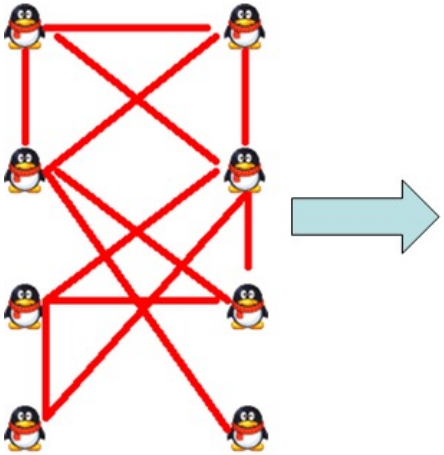
中介者模式概述

- 联合国



中介者模式概述

- QQ聊天示意图



(A)

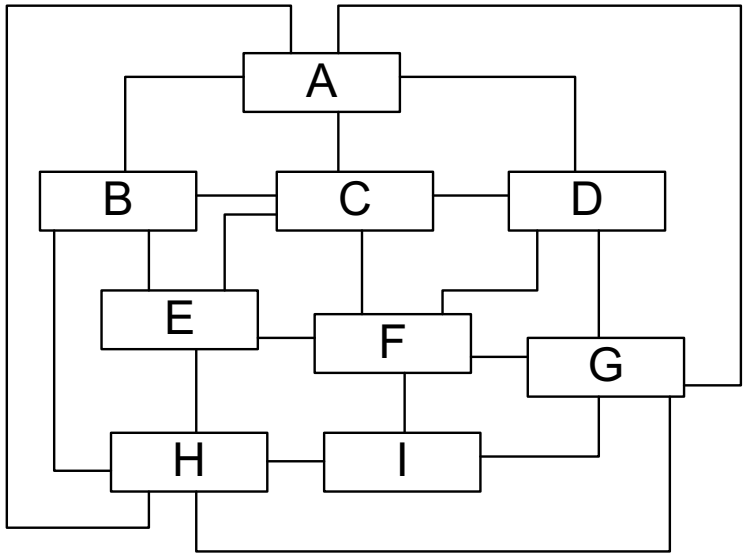
(B)

中介者模式概述

- 分析
  - QQ聊天的两种方式：
    - (1) **用户与用户直接聊天**，用户与用户之间存在多对多的联系，这将导致系统中用户之间的关系非常复杂，一个用户如果要将相同的信息或文件发送给其他所有用户，必须一个一个地发送
    - (2) **通过QQ群聊天**，用户只需要将信息或文件发送到群中或上传为群共享文件即可，群的作用就是将发送者所发送的信息和文件转发给每一个接收者，将极大地减少系统中用户之间的两两通信

中介者模式概述

- 分析
  - 软件开发：
    - **网状结构**：多对多联系将导致系统非常复杂，几乎每个对象都需要与其他对象发生相互作用，而这种相互作用表现为一个对象与另外一个对象的直接耦合，这将导致一个过度耦合的系统

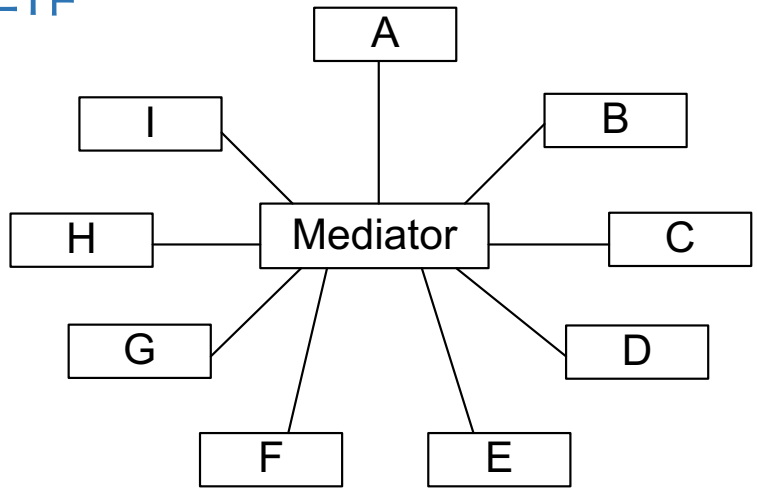


网状结构

中介者模式概述

- 分析
  - 软件开发：
    - **星型结构**：中介者模式将系统的网状结构变成以中介者为中心的星型结构，同事对象不再直接与另一个对象联系，它通过中介者对象与另一个对象发生相互作用。系统的结构不会因为新对象的引入带来大量的修改工作

中介者模式



星型结构

中介者模式概述

- 中介者模式的定义

中介者模式：定义一个对象来封装一系列对象的交互。中介者模式使各对象之间不需要显式地相互引用，从而使其耦合松散，而且让你可以独立地改变它们之间的交互。

Mediator Pattern: Define an object that encapsulates how a set of objects interact. Mediator promotes **loose coupling** by keeping objects from referring to each other explicitly, and **it lets you vary their interaction independently**.

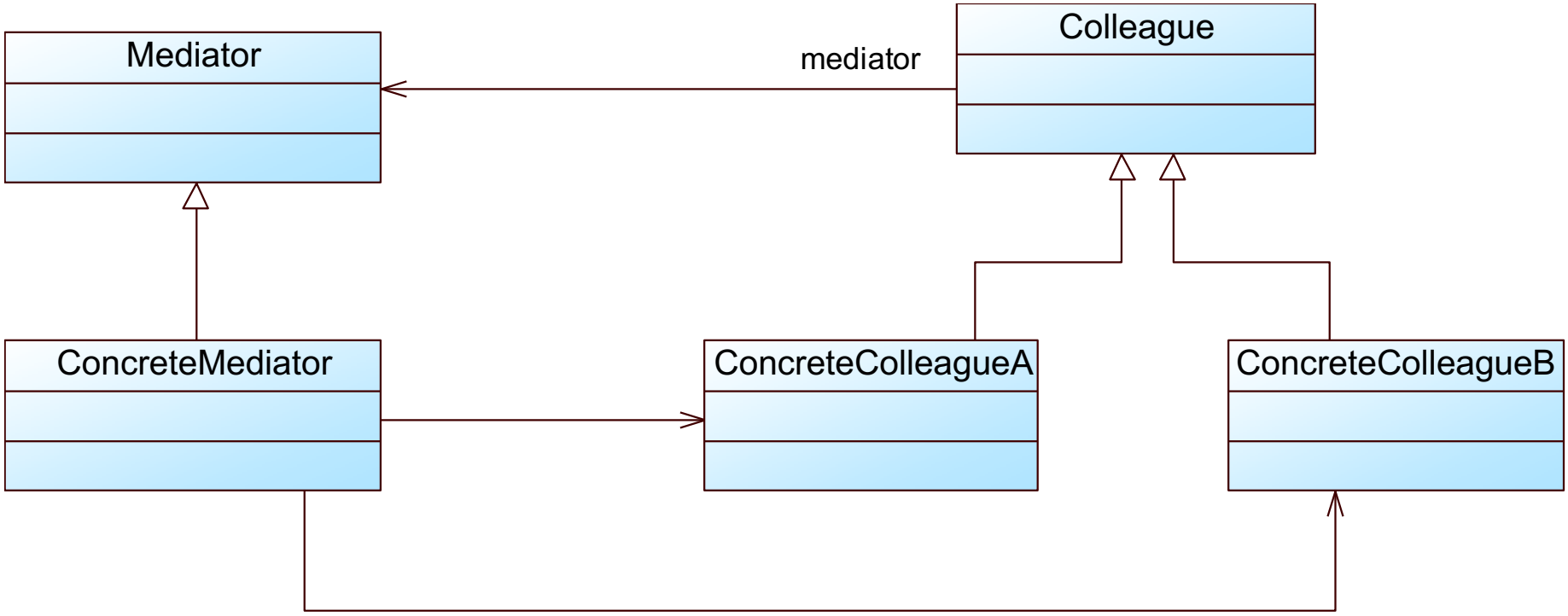
- 对象行为型模式

中介者模式概述

- 中介者模式的定义
  - 又称为调停者模式
  - 在中介者模式中，通过引入中介者来简化对象之间的复杂交互
  - 中介者模式是迪米特法则的一个典型应用
  - 对象之间多对多的复杂关系转化为相对简单的一对多关系

中介者模式的结构与实现

- 中介者模式的结构



中介者模式的结构与实现

- 中介者模式的结构
  - 中介者模式包含以下4个角色：
    - Mediator（抽象中介者）
    - ConcreteMediator（具体中介者）
    - Colleague（抽象同事类）
    - ConcreteColleague（具体同事类）

中介者模式的结构与实现

- 中介者模式的实现
  - 中介者类的职责
    - **中转作用（结构性）**：各个同事对象不再需要显式地引用其他同事，当需要和其他同事进行通信时，可通过中介者来实现间接调用
    - **协调作用（行为性）**：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装

中介者模式的结构与实现

- 中介者模式的实现
  - 典型的抽象中介者类代码：

```
public abstract class Mediator {
    protected ArrayList<Colleague> colleagues = new
ArrayList<Colleague>(); //用于存储同事对象

    //注册方法，用于增加同事对象
    public void register(Colleague colleague) {
        colleagues.add(colleague);
    }

    //声明抽象的业务方法
    public abstract void operation();
}
```

中介者模式的结构与实现

- 中介者模式的实现
  - 典型的具体中介者类代码：

```
public class ConcreteMediator extends Mediator {
    //实现业务方法，封装同事之间的调用
    public void operation() {
        ......
        ((Colleague)(colleagues.get(0))).method1(); //通过中介者调用同事类的方法
        ......
    }
}
```

中介者模式的结构与实现

- 中介者模式的实现
  - 典型的抽象同事类代码：

```
public abstract class Colleague {
    protected Mediator mediator; //维持一个抽象中介者的引用

    public Colleague(Mediator mediator) {
        this.mediator=mediator;
    }

    public abstract void method1(); //声明自身方法，处理自己的行为

    //定义依赖方法，与中介者进行通信
    public void method2() {
        mediator.operation();
    }
}
```

中介者模式的结构与实现

- 中介者模式的实现
  - 典型的具体同事类代码：

```
public class ConcreteColleague extends Colleague {
    public ConcreteColleague(Mediator mediator) {
        super(mediator);
    }

    //实现自身方法
    public void method1() {
        ......
    }
}
```

某软件公司要开发一套CRM系统，其中包含一个客户信息管理模块，所设计的“客户信息管理窗口”界面效果图如下图所示：

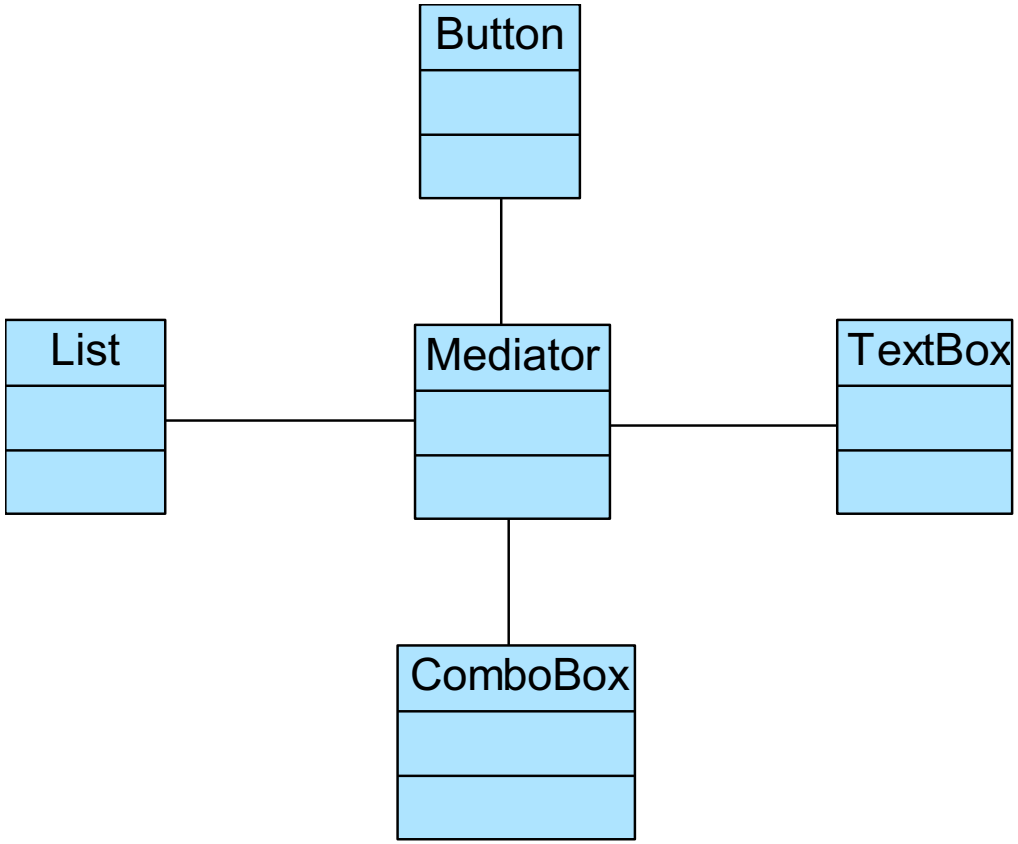
“客户信息管理窗口”界面效果图

通过分析发现，在上图中，界面组件之间存在较为复杂的交互关系：如果删除一个客户，将从客户列表(List)中删掉对应的项，客户选择组合框(ComboBox)中的客户名称也将减少一个；如果增加一个客户信息，则客户列表中将增加一个客户，且组合框中也将增加一项。

为了更好地处理界面组件之间的交互，现使用中介者模式设计该系统。

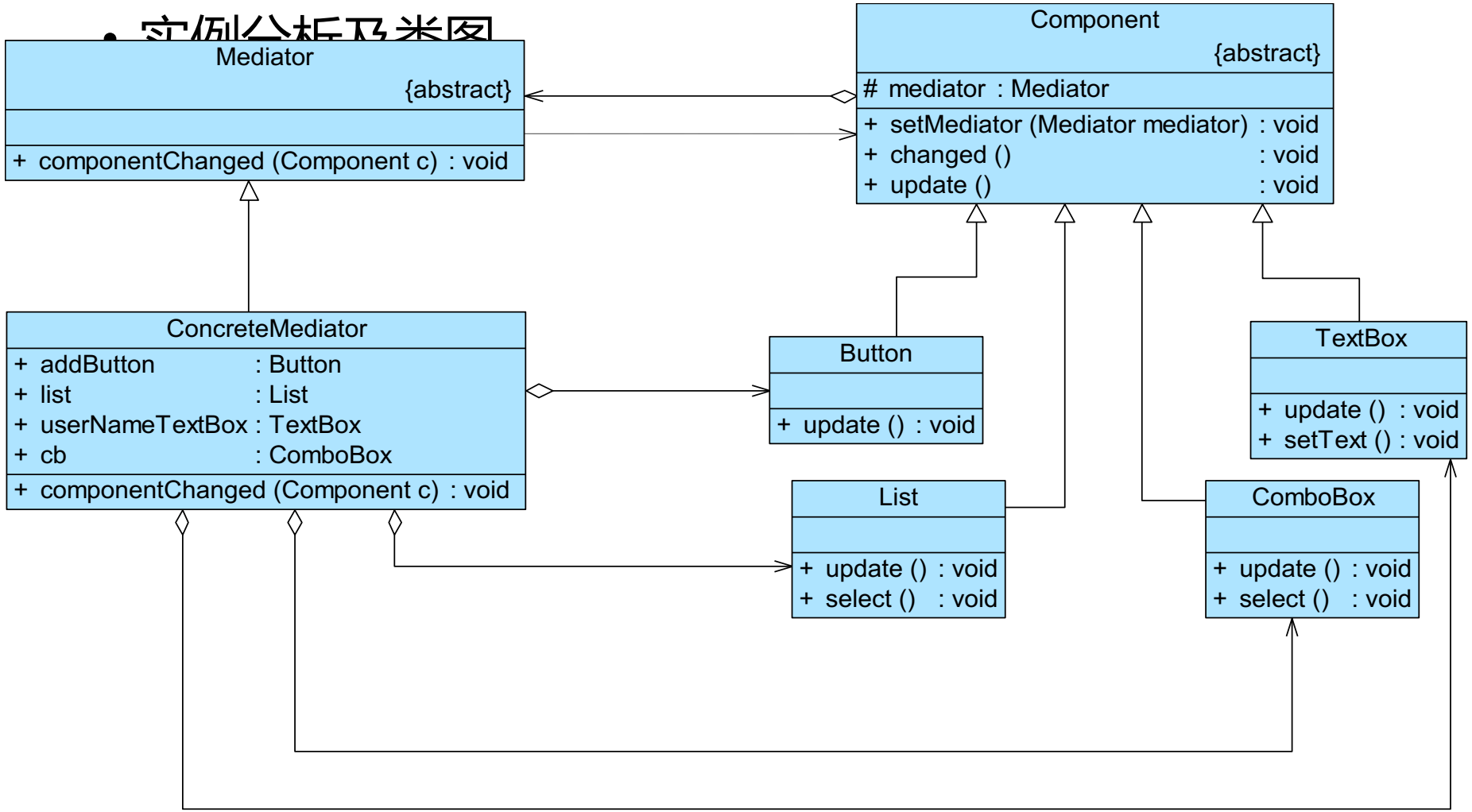
中介者模式的应用实例

- 实例分析及类图



引入了中介者类的“客户信息管理窗口”结构示意图

中介者模式的应用实例



“客户信息管理窗口”结构图

中介者模式的应用实例

- 实例代码
  - (1) Mediator：抽象中介者类
  - (2) ConcreteMediator：具体中介者类
  - (3) Component：抽象组件类，充当抽象同事类
  - (4) Button：按钮类，充当具体同事类
  - (5) List：列表框类，充当具体同事类
  - (6) ComboBox：组合框类，充当具体同事类
  - (7) TextBox：文本框类，充当具体同事类
  - (8) Client：客户端测试类

演示……

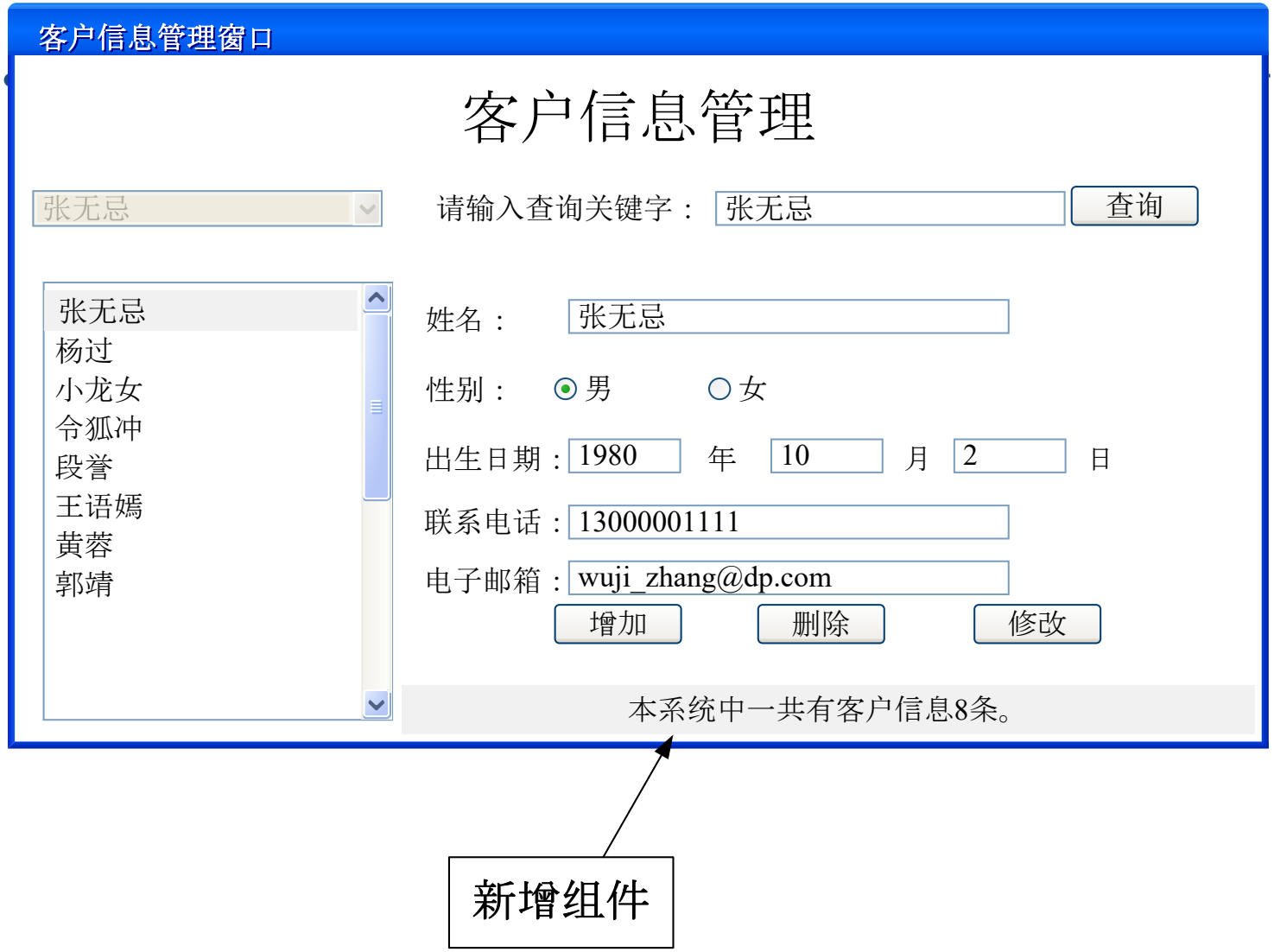
Code (designpatterns.mediator)

中介者模式的应用实例

- 结果及分析
  - 当某个组件类的changed()方法被调用时，中介者的componentChanged()方法将被调用，在中介者的componentChanged()方法中再逐个调用与该组件有交互的其他组件的相关方法
  - 如果某个组件类需要与新的组件进行交互，无须修改已有组件类的源代码，只需修改中介者或者对现有中介者进行扩展即可，系统具有更好的灵活性和可扩展性

扩展中介者与同事类

- 目的



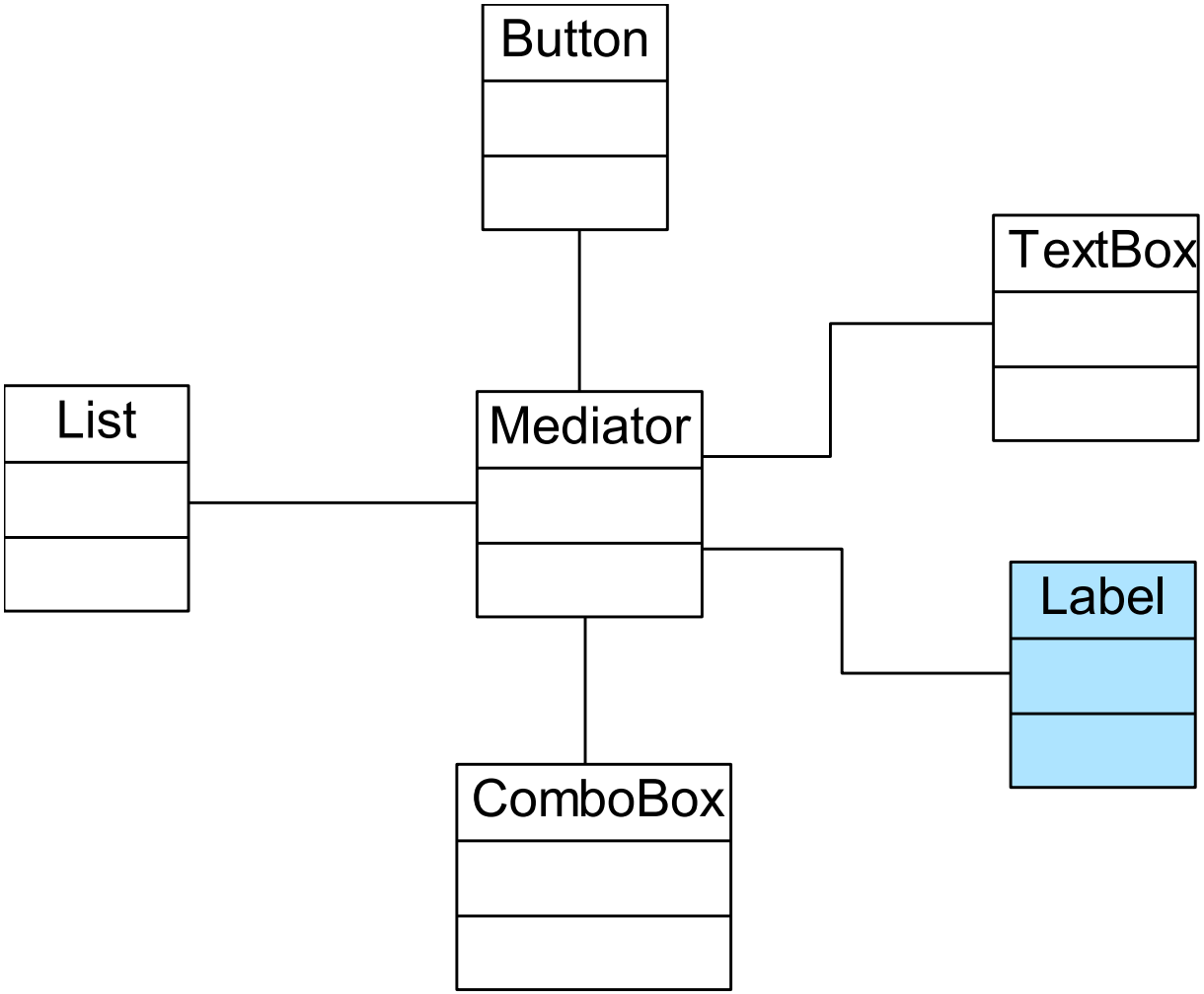
扩展中介者与同事类

- 解决方案
  - (1) 增加一个界面组件类Label，修改原有的具体中介者类ConcreteMediator，增加一个对Label对象的引用
  - (2) 增加一个界面组件类Label，增加一个ConcreteMediator的子类SubConcreteMediator来实现对Label对象的引用

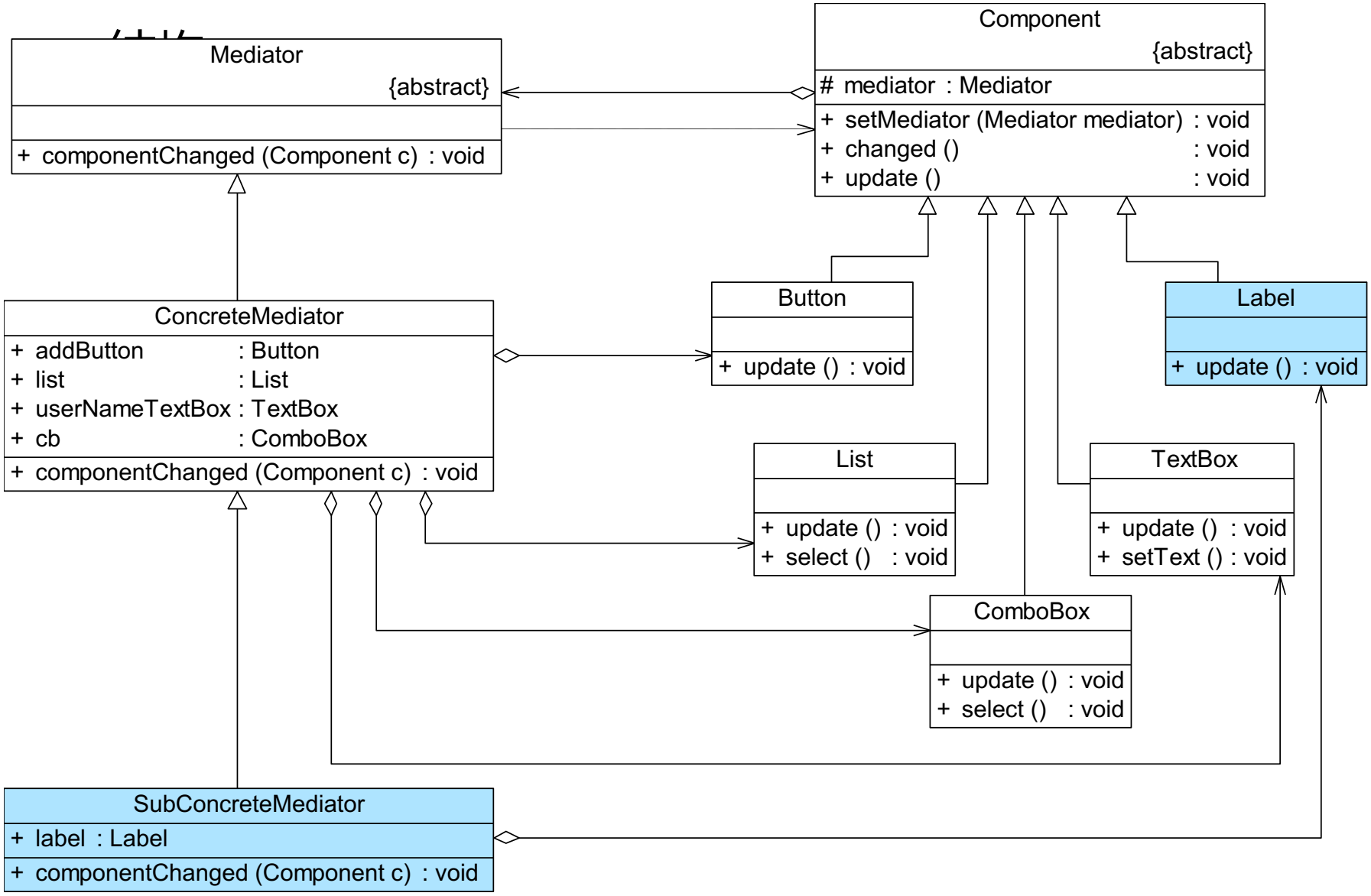
方案(2)更符合开闭原则

扩展中介者与同事类

- 结构



扩展中介者与同事类


Mediator
{abstract}
+ componentChanged (Component c) : void
Component
{abstract}
mediator : Mediator
+ setMediator (Mediator mediator) : void
+ changed () : void
+ update () : void
ConcreteMediator
+ addButton : Button
+ list : List
+ userNameTextBox : TextBox
+ cb : ComboBox
+ componentChanged (Component c) : void
Button
+ update () : void
Label
+ update () : void
List
+ update () : void
+ select () : void
TextBox
+ update () : void
+ setText () : void
ComboBox
+ update () : void
+ select () : void
SubConcreteMediator
+ label : Label
+ componentChanged (Component c) : void

中介者模式的优缺点与适用环境

- 模式优点
  - 简化了对象之间的交互，它用中介者和同事的一对多交互代替了原来同事之间的多对多交互，将原本难以理解的网状结构转换成相对简单的星型结构
  - 可将各同事对象解耦
  - 可以减少子类生成，中介者模式将原本分布于多个对象间的行为集中在一起，改变这些行为只需生成新的中介者子类即可，这使得各个同事类可被重用，无须直接对同事类进行扩展

中介者模式的优缺点与适用环境

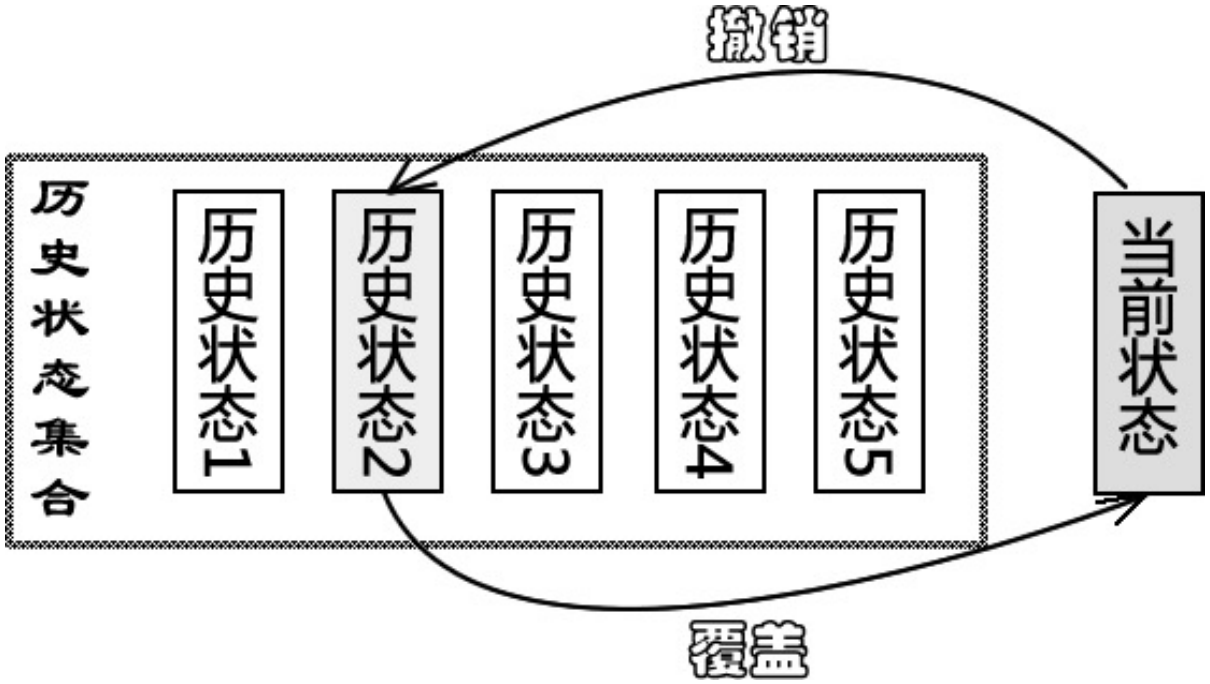
- 模式缺点
  - 在具体中介者类中包含了大量的同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护

中介者模式的优缺点与适用环境

- 模式适用环境
  - 系统中对象之间存在复杂的引用关系，系统结构混乱且难以理解
  - 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象
  - 想通过一个中间类来封装多个类中的行为，又不想生成太多的子类

备忘录模式概述

- 备忘录模式——软件中的“后悔药”——撤销(Undo)



备忘录模式概述

- 分析
  - 通过使用备忘录模式可以让系统恢复到某一特定的历史状态
  - 首先保存软件系统的历史状态，当用户需要取消错误操作并且返回到某个历史状态时，可以取出事先保存的历史状态来覆盖当前状态

备忘录模式概述

- 备忘录模式的定义

备忘录模式：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样就可以在以后将对象恢复到原先保存的状态。

Memento Pattern: **Without violating encapsulation**, capture and externalize an object's internal state so that **the object can be restored to this state later**.

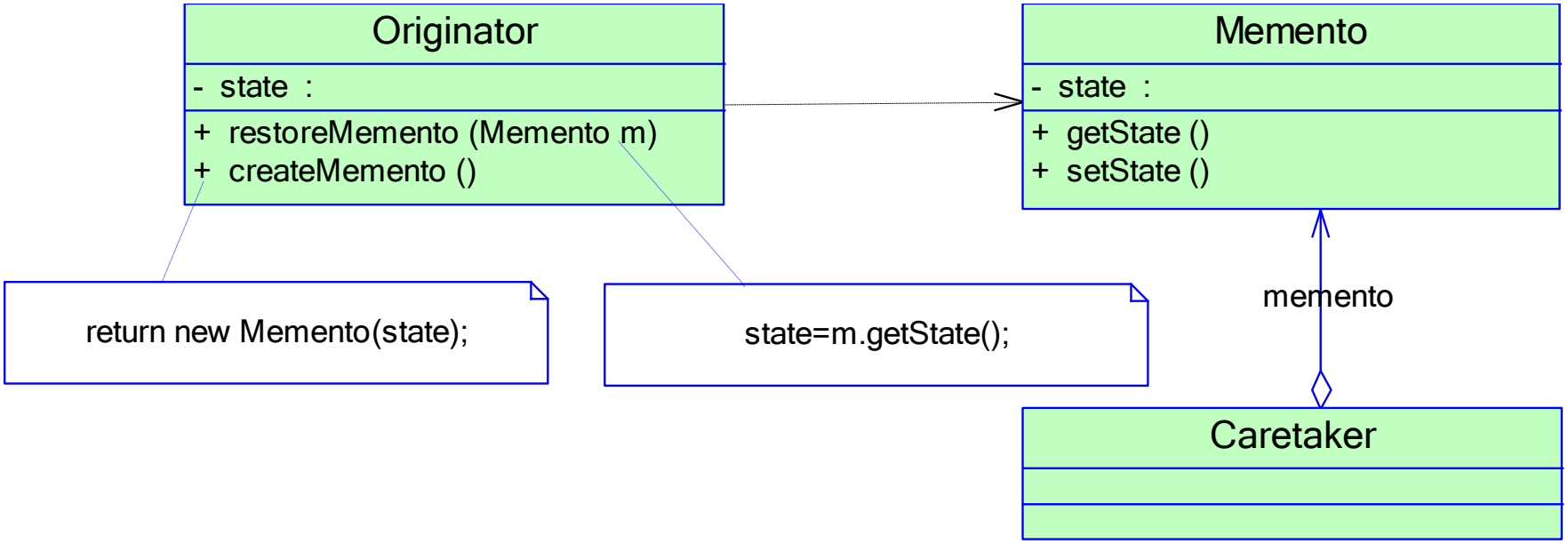
- 对象行为型模式

备忘录模式概述

- 备忘录模式的定义
  - 别名为标记(Token)模式
  - 提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤
  - 当前在很多软件所提供的撤销(Undo)操作中就使用了备忘录模式

备忘录模式的结构与实现

- 备忘录模式的结构



备忘录模式的结构与实现

- 备忘录模式的结构
  - 备忘录模式包含以下3个角色：
    - Originator（原发器）
    - Memento（备忘录)
    - Caretaker（负责人）

```
package designpatterns.memento;
public class Originator {
    private String state;

    public Originator(){}

    //创建一个备忘录对象
    public Memento createMemento() {
        return new Memento(this);
    }

    //根据备忘录对象恢复原发器状态
    public void restoreMemento(Memento m) {
        state = m.state;
    }

    public void setState(String state) {
        this.state=state;
    }

    public String getState() {
        return this.state;
    }
}
```

备忘录模式的结构与实现

```
package designpatterns.memento;

//备忘录类，默认可见性，包内可见
class Memento {
    private String state;

    Memento(Originator o) {
        state = o.getState();
    }

    void setState(String state) {
        this.state=state;
    }

    String getState() {
        return this.state;
    }
}
```

备忘录模式的结构与实现

- 备忘录模式的实现
  - 除了Originator类，不允许其他类来调用备忘录类Memento的构造函数与相关方法
  - 如果允许其他类调用setState()等方法，将导致在备忘录中保存的历史状态发生改变，通过撤销操作所恢复的状态就不再是真实的历史状态，备忘录模式也就失去了本身的意义
  - 理想的情况是只允许生成该备忘录的原发器访问备忘录的内部状态

备忘录模式的结构与实现

- 备忘录模式的实现
  - Java语言实现：
    - 将Memento类与Originator类定义在同一个包(package)中来实现封装，使用默认可见性定义Memento类，即保证其在包内可见
    - 将备忘录类作为原发器类的内部类，使得只有原发器才可以访问备忘录中的数据，其他对象都无法使用备忘录中的数据

备忘录模式的结构与实现

- 备忘录模式的实现
  - 典型的负责人类代码：

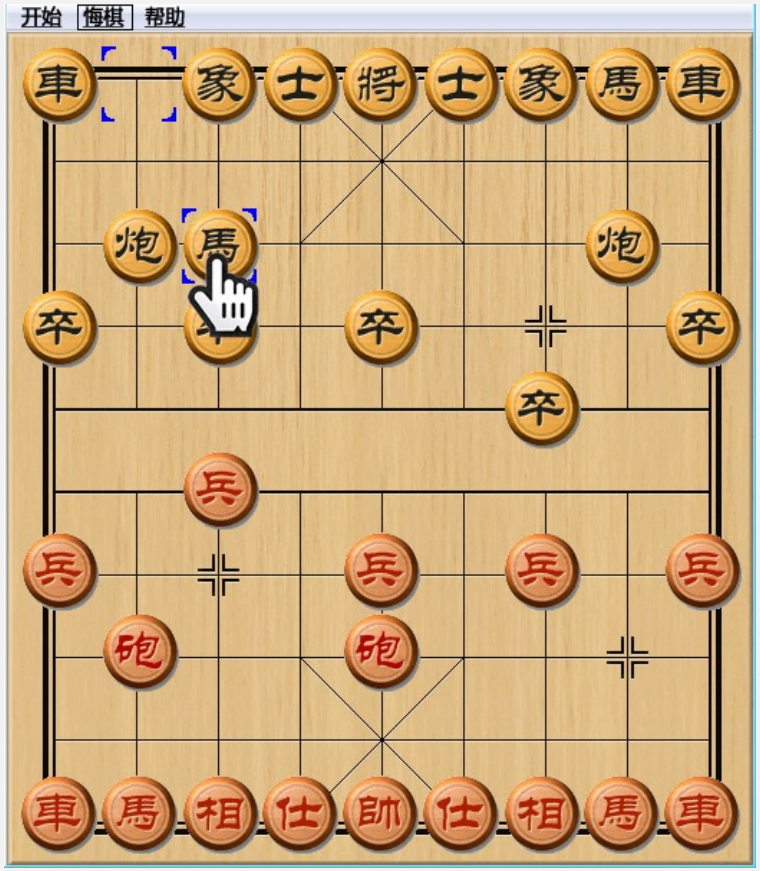
```
package designpatterns.memento;

public class Caretaker {
    private Memento memento;

    public Memento getMemento() {
        return memento;
    }

    public void setMemento(Memento memento) {
        this.memento=memento;
    }
}
```

某软件公司要使用Java语言开发一款可以运行在Android平台的触摸式中国象棋软件，由于考虑到有些用户是“菜鸟”，经常不小心走错棋；还有些用户因为不习惯使用手指在手机屏幕上拖动棋子，常常出现操作失误，因此该中国象棋软件要提供“悔棋”功能，在用户走错棋或操作失误后可恢复到前一个步骤。如下图所示：

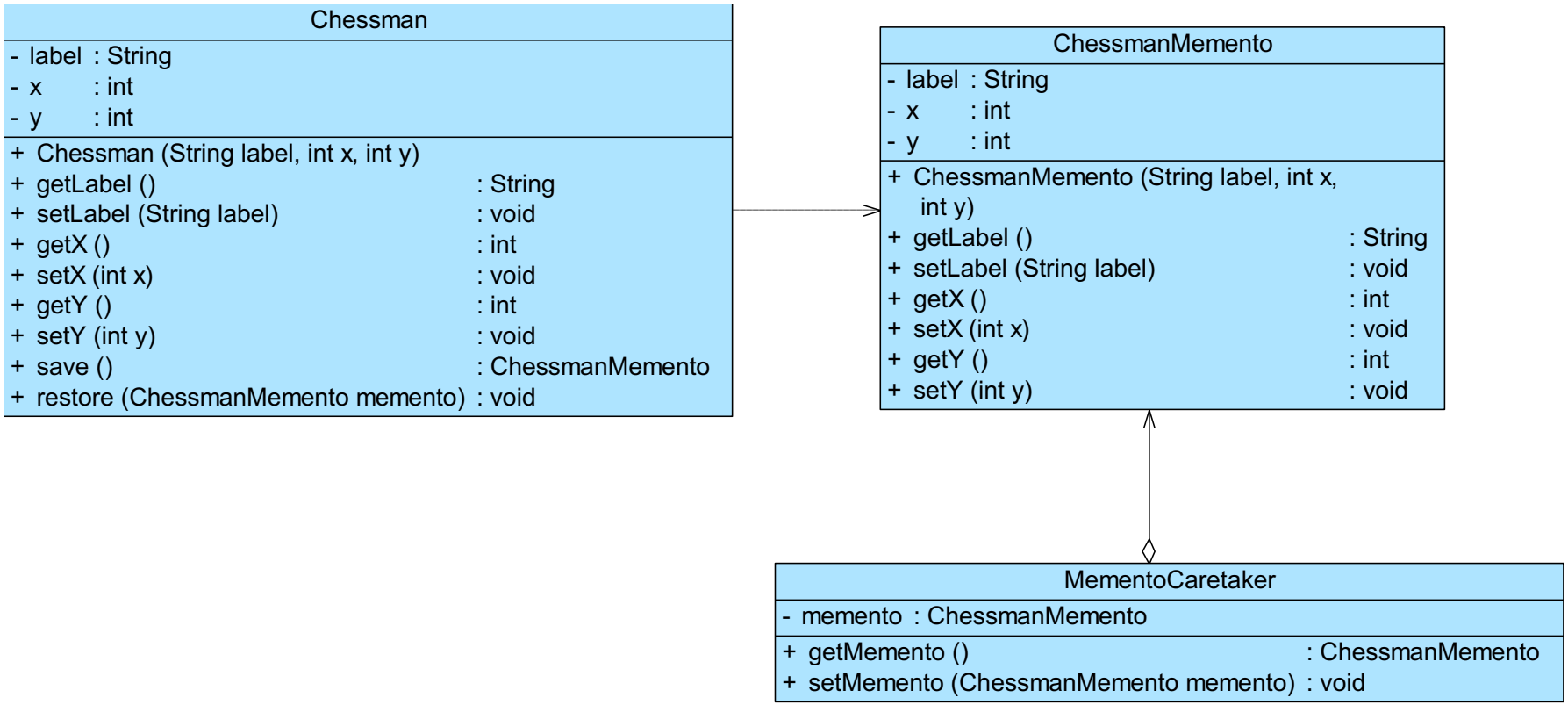


中国象棋软件界面示意图

为了实现“悔棋”功能，现使用备忘录模式来设计该中国象棋软件。

备忘录模式的应用实例

- 实例类图



中国象棋棋子撤销功能结构图

备忘录模式的应用实例

- 实例代码
  - (1) Chessman：象棋棋子类，充当原发器
  - (2) ChessmanMemento：象棋棋子备忘录类，充当备忘录
  - (3) MementoCaretaker：象棋棋子备忘录管理类，充当负责人
  - (4) Client：客户端测试类

演示……

Code (designpatterns.memento)

备忘录模式的应用实例

- 结果及分析
  - 通过创建备忘录对象可以将象棋棋子的历史状态信息记录下来，在“悔棋”时取出存储在备忘录中的历史状态信息，用历史状态来覆盖当前状态，从而实现状态的撤销

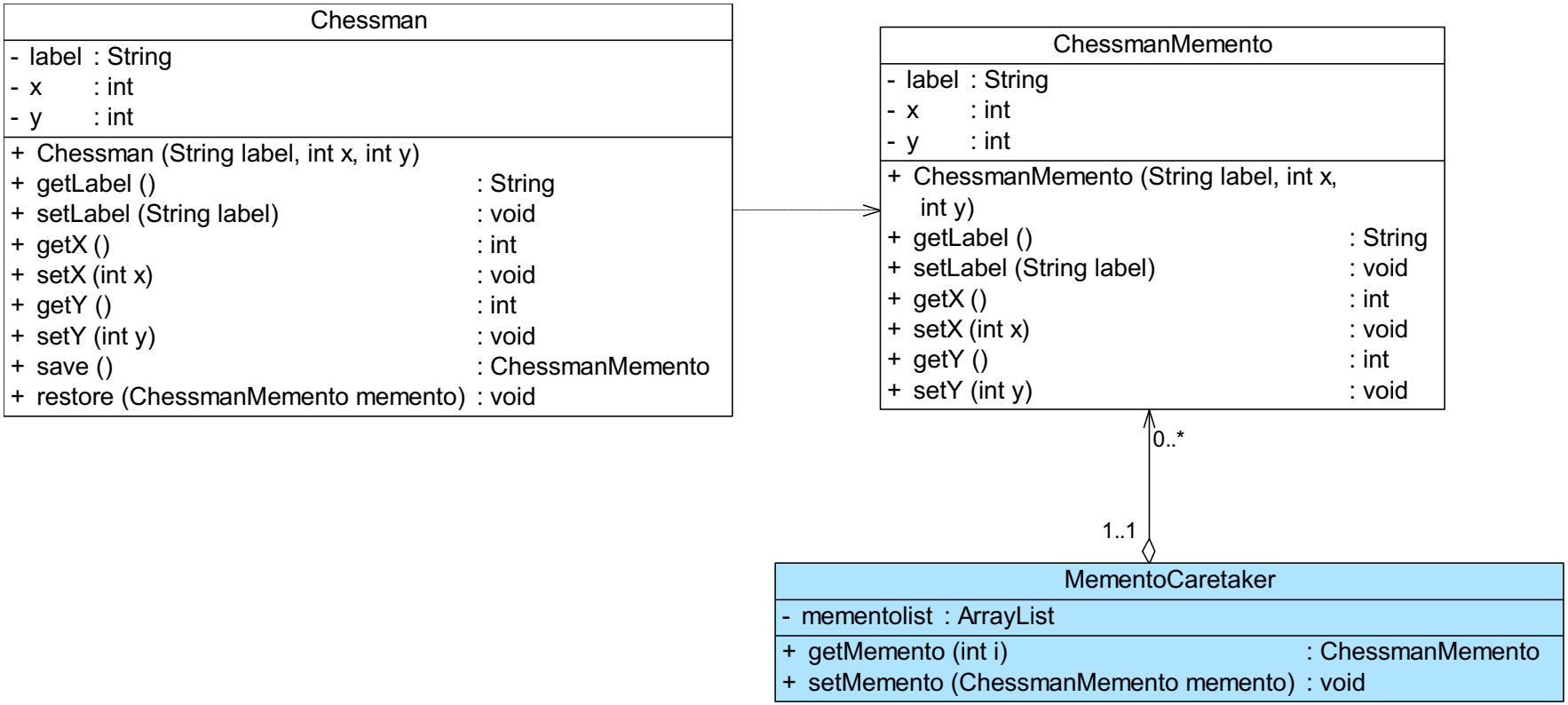
```
棋子车当前位置为：第1行第1列。
棋子车当前位置为：第1行第4列。
棋子车当前位置为：第5行第4列。
******悔棋******
棋子车当前位置为：第1行第4列。
```

实现多次撤销

- 动机
  - 有时候用户需要撤销多步操作
  - 实现方案：在负责人类中定义一个集合来存储多个备忘录，每个备忘录负责保存一个历史状态，在撤销时可以对备忘录集合进行逆向遍历，回到一个指定的历史状态，还可以对备忘录集合进行正向遍历，实现重做(Redo)或恢复操作，即取消撤销，让对象状态得到恢复

实现多次撤销

- 结构



改进之后的中国象棋棋子撤销功能结构图

实现多次撤销

- 实现

```
import java.util.*;

public class MementoCaretaker {
   //定义一个集合来存储多个备忘录
   private ArrayList<ChessmanMemento> mementolist = new ArrayList
<ChessmanMemento>();

   public ChessmanMemento getMemento(int i) {
      return (ChessmanMemento)mementolist.get(i);
   }

   public void setMemento(ChessmanMemento memento) {
      mementolist.add(memento);
   }
}
```

备忘录模式的优缺点与适用环境

- 模式优点
  - 提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤
  - 实现了对信息的封装，一个备忘录对象是一种原发器对象状态的表示，不会被其他代码所改动

备忘录模式的优缺点与适用环境

- 模式缺点
  - 资源消耗过大，如果需要保存的原发器类的成员变量太多，就不可避免地需要占用大量的存储空间，每保存一次对象的状态都需要消耗一定的系统资源

备忘录模式的优缺点与适用环境

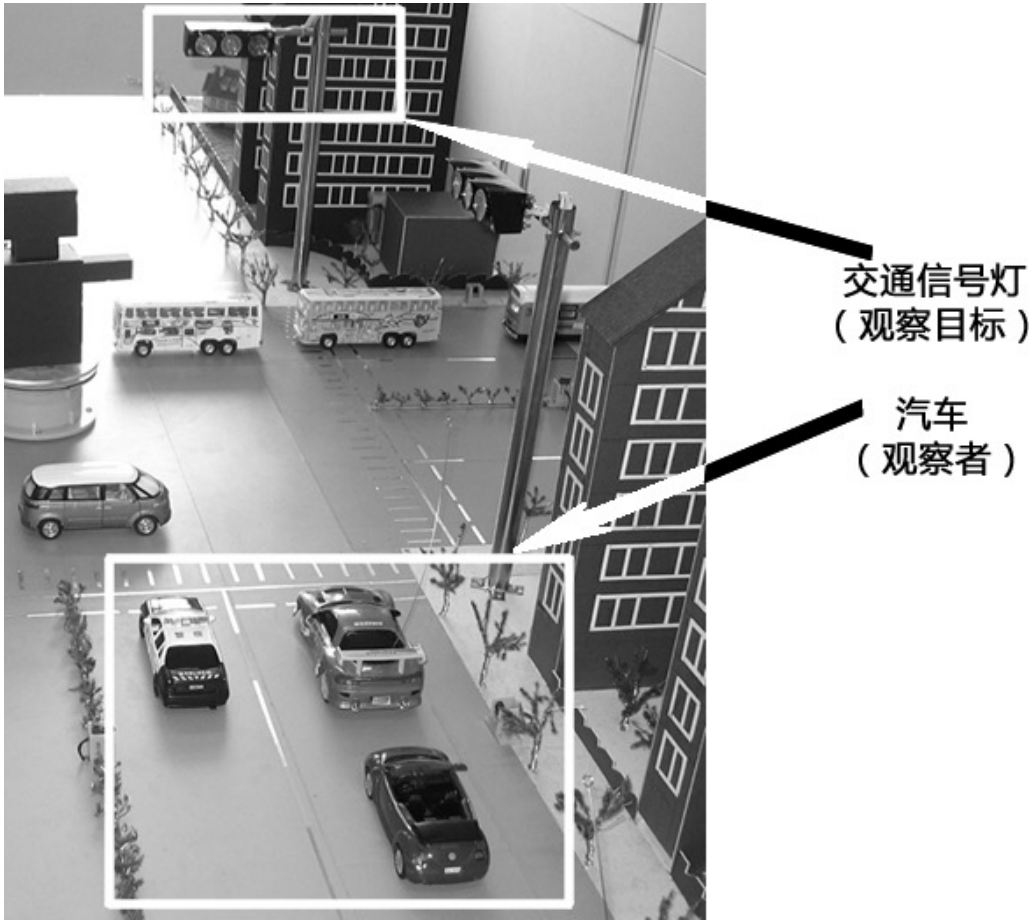
- 模式适用环境
  - 保存一个对象在某一个时刻的全部状态或部分状态，这样以后需要时能够恢复到先前的状态，实现撤销操作
  - 防止外界对象破坏一个对象历史状态的封装性，避免将对象历史状态的实现细节暴露给外界对象

思考

- 如何使用栈(Stack)实现多步撤销(Undo)和重做(Redo)操作？

观察者模式概述

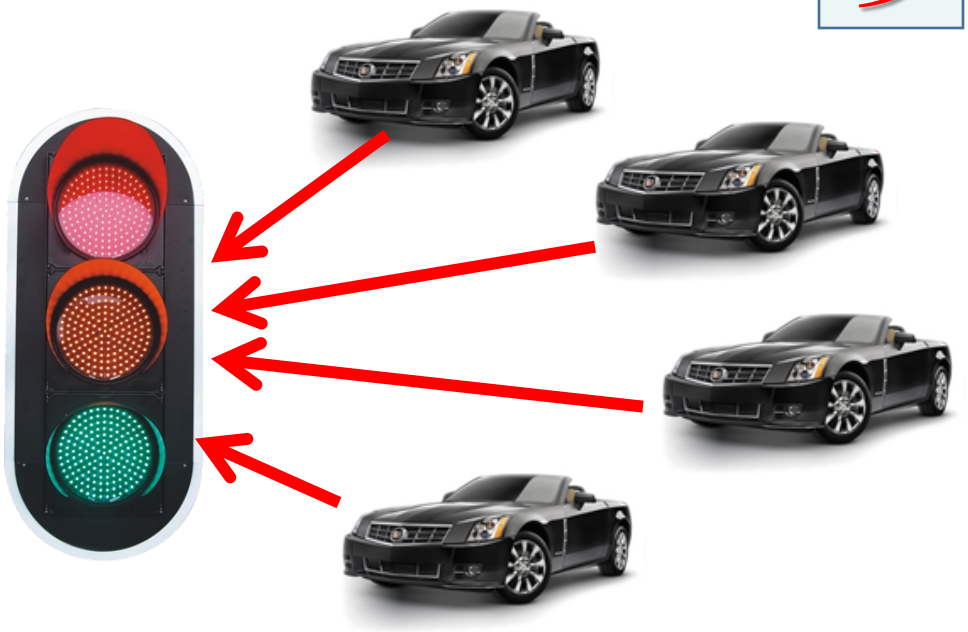
- 交通信号灯与汽车示意图



观察者模式概述

- 分析
  - 交通信号灯 ← → 观察目标
  - 汽车（汽车驾驶员）← →观察者

一对多



观察者模式概述

- 分析
  - 软件系统：一个对象的状态或行为的变化将导致其他对象的状态或行为也发生改变，它们之间将产生联动
  - 观察者模式：
    - 定义了对象之间一种一对多的依赖关系，让一个对象的改变能够影响其他对象
    - 发生改变的对象称为观察目标，被通知的对象称为观察者
    - 一个观察目标可以对应多个观察者

观察者模式概述

- 观察者模式的定义

观察者模式：定义对象之间的一种**一对多依赖关系**，使得每当**一个对象状态发生改变**时，其相关依赖对象**都得到通知并被自动更新**。

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**.

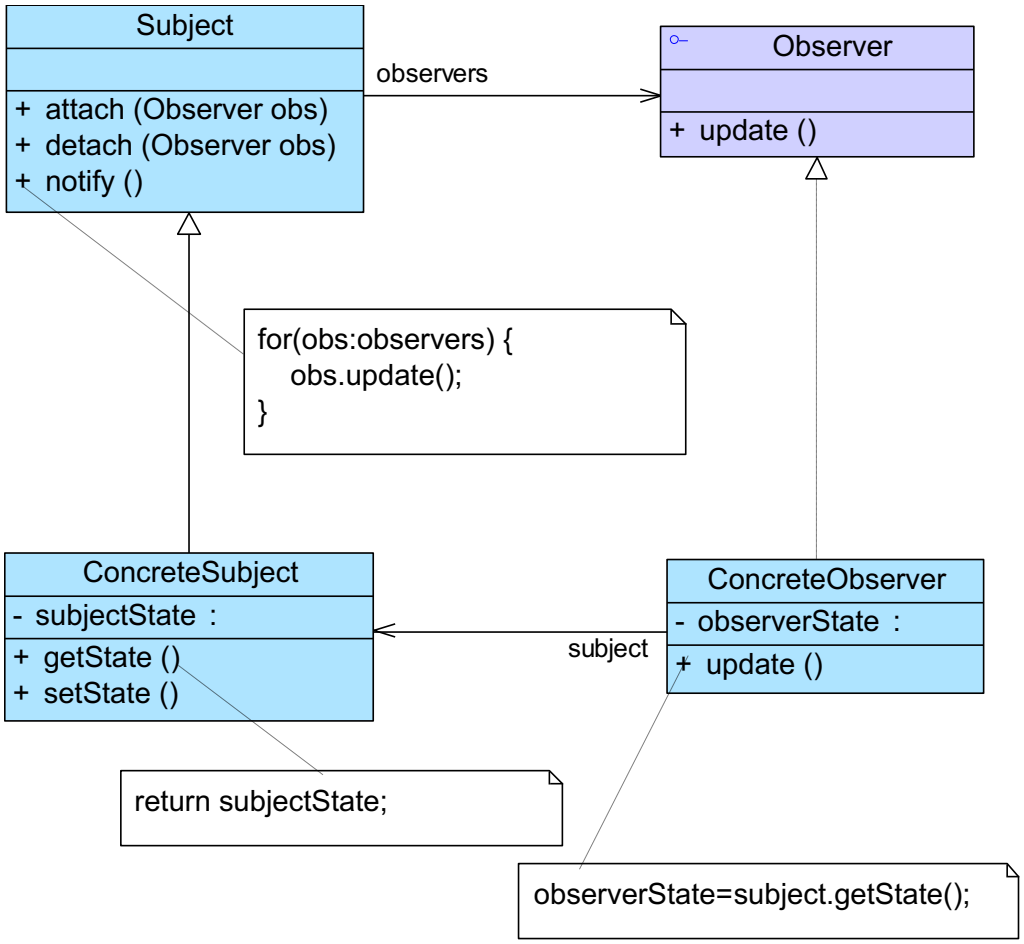
- 对象行为型模式

观察者模式概述

- 观察者模式的定义
  - 别名
    - 发布-订阅(Publish/Subscribe)模式
    - 模型-视图(Model/View)模式
    - 源-监听器(Source/Listener)模式
    - 从属者(Dependents)模式

观察者模式的结构与实现

- 观察者模式的结构



观察者模式的结构与实现

- 观察者模式的结构
  - 观察者模式包含以下4个角色：
    - Subject（目标）
    - ConcreteSubject（具体目标）
    - Observer（观察者）
    - ConcreteObserver（具体观察者）

观察者模式的结构与实现

```
import java.util.*;
public abstract class Subject {
    //定义一个观察者集合用于存储所有观察者对象
    protected ArrayList observers<Observer> = new ArrayList();

    //注册方法，用于向观察者集合中增加一个观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }

    //注销方法，用于在观察者集合中删除一个观察者
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    //声明抽象通知方法
    public abstract void notify();
}
```

观察者模式的结构与实现

- 观察者模式的实现
  - 典型的具体目标类代码：

```
public class ConcreteSubject extends Subject {
    //实现通知方法
    public void notify() {
        //遍历观察者集合，调用每一个观察者的响应方法
        for(Object obs:observers) {
            ((Observer)obs).update();
        }
    }
}
```

观察者模式的结构与实现

- 观察者模式的实现
  - 典型的抽象观察者代码：

```
public interface Observer {
    //声明响应方法
    public void update();
}
```

观察者模式的结构与实现

- 观察者模式的实现
  - 典型的具体观察者代码：

```
public class ConcreteObserver implements Observer {
    //实现响应方法
    public void update() {
        //具体响应代码
    }
}
```

观察者模式的结构与实现

- 观察者模式的实现
  - 说明：
    - 有时候在具体观察者类ConcreteObserver中需要使用到具体目标类ConcreteSubject中的状态（属性），会存在关联或依赖关系
    - 如果在具体层之间具有关联关系，系统的扩展性将受到一定的影响，增加新的具体目标类有时候需要修改原有观察者的代码，在一定程度上违背了开闭原则，但是如果原有观察者类无须关联新增的具体目标，则系统扩展性不受影响

观察者模式的结构与实现

- 观察者模式的实现
  - 典型的客户端代码片段：

```
……
Subject subject = new ConcreteSubject();
Observer observer = new ConcreteObserver();
subject.attach(observer); //注册观察者
subject.notify();
……
```

观察者模式的应用实例

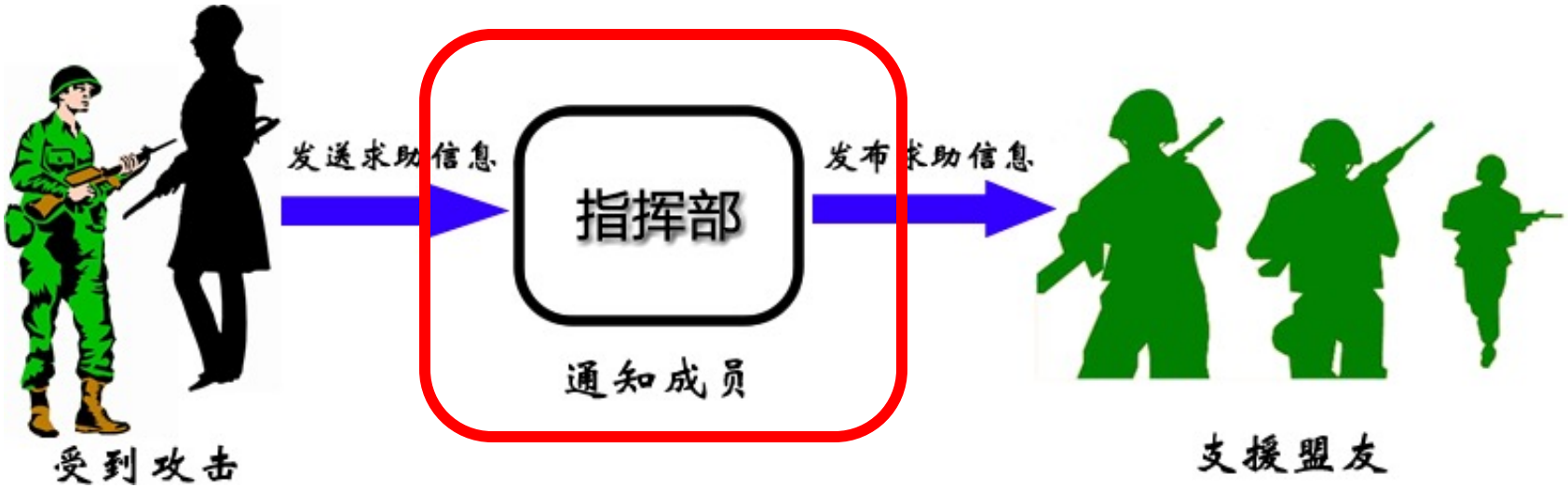
- 实例说明

在某多人联机对战游戏中，多个玩家可以加入同一战队组成联盟，当战队中的某一成员受到敌人攻击时将给所有其他盟友发送通知，盟友收到通知后将做出响应。

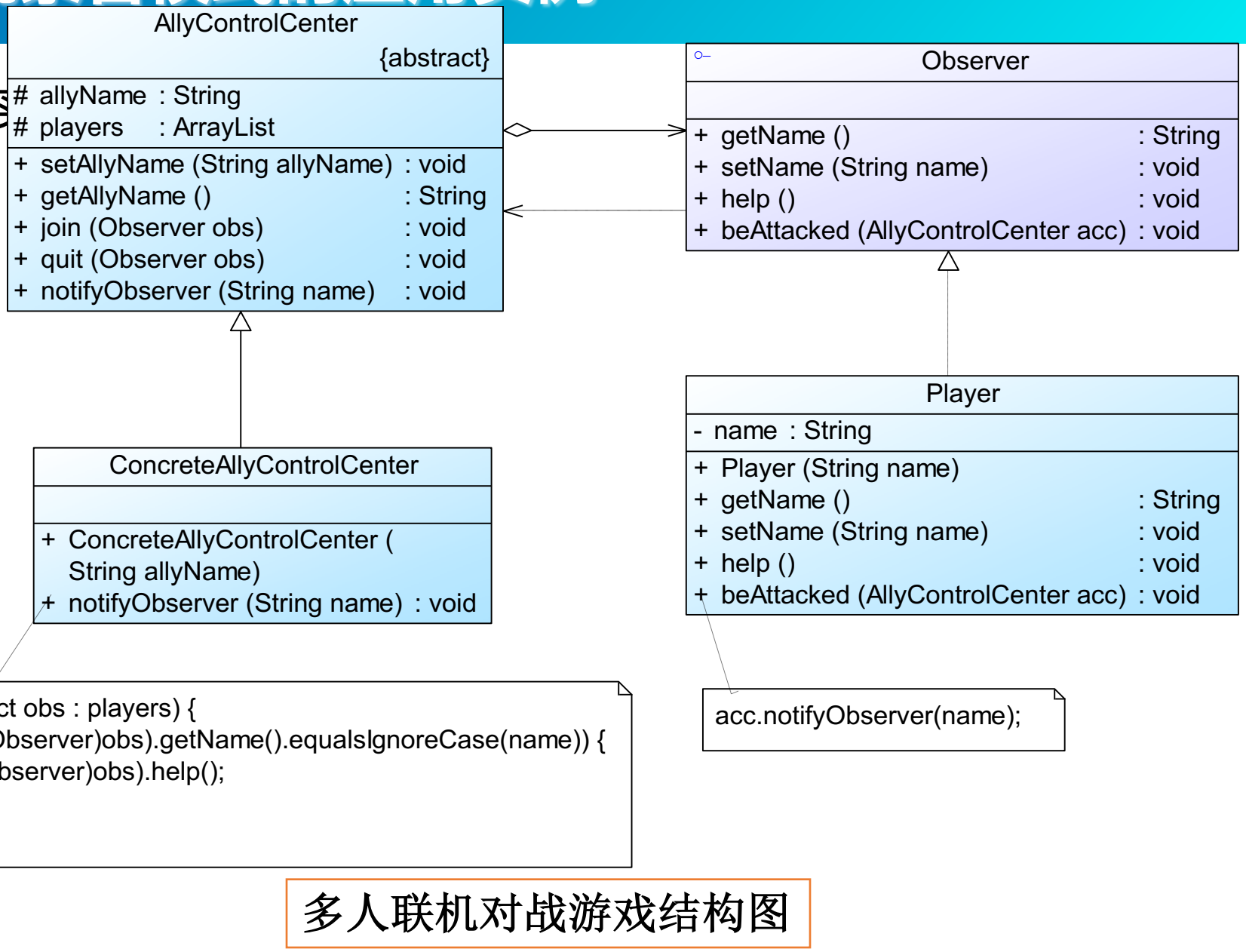
现使用观察者模式设计并实现该过程，以实现战队成员之间的联动。

观察者模式的应用实例

- 实例分析及类图
  - 战队成员之间的联动过程：
    - 联盟成员受到攻击 → 发送通知给盟友 → 盟友做出响应



观察者模式的应用实例



多人联机对战游戏结构图

观察者模式的应用实例

- 实例代码
  - (1) AllyControlCenter：指挥部（战队控制中心）类，充当抽象目标类
  - (2) ConcreteAllyControlCenter：具体指挥部类，充当具体目标类
  - (3) Observer：抽象观察者类
  - (4) Player：战队成员类，充当具体观察者类
  - (5) Client：客户端测试类

演示.....

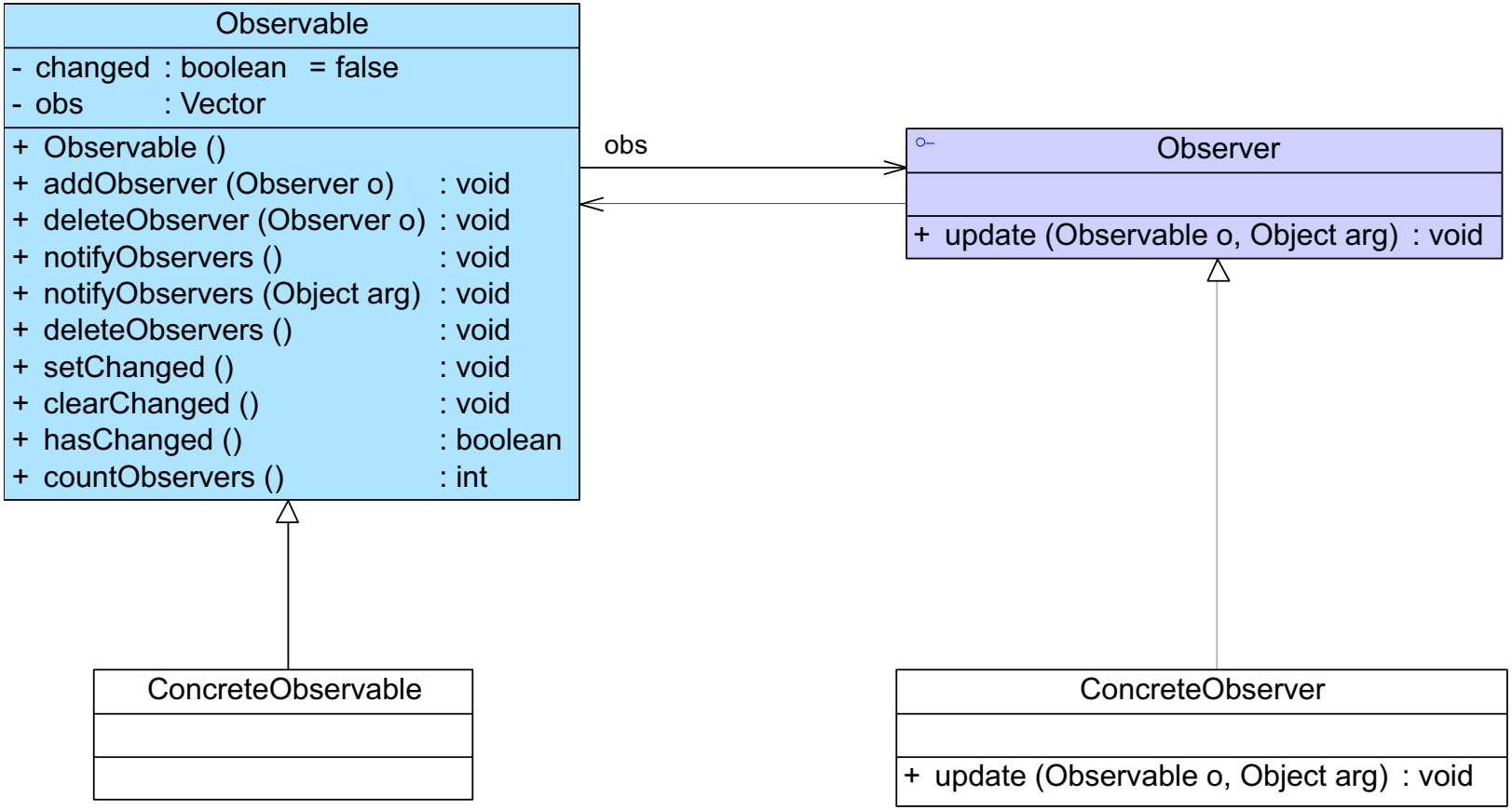
Code (designpatterns.observer)

观察者模式的应用实例

- 结果及分析
  - 两次对象之间的联动，触发链： Player.beAttacked() → AllyControlCenter.notifyObserver() → Player.help()

JDK对观察者模式的支持

- java.util.Observer
- java.util.Observable



观察者模式与Java事件处理

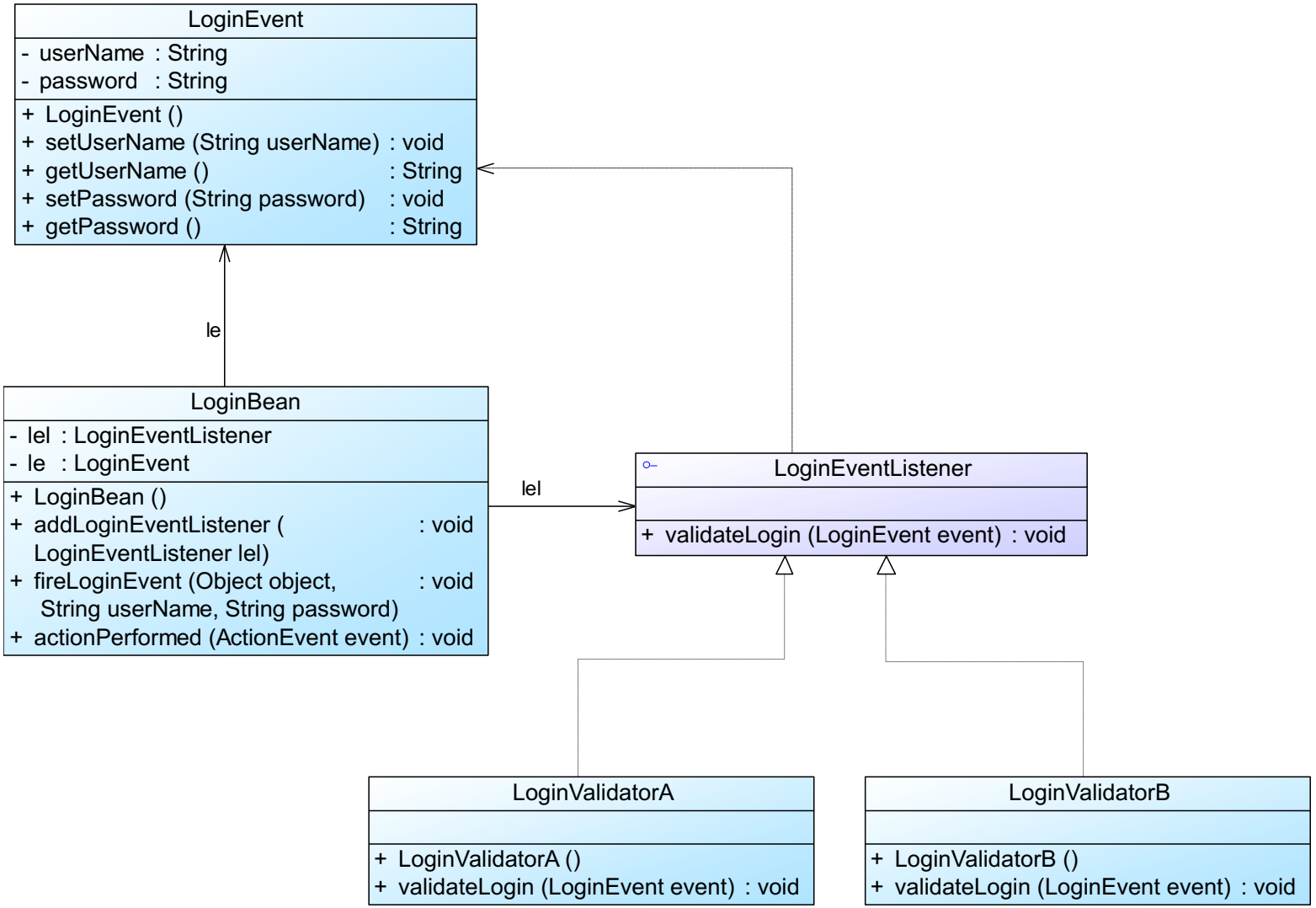
- 分析
  - 事件源对象充当观察目标角色，事件监听器充当抽象观察者角色，事件处理对象充当具体观察者角色
  - 如果事件源对象的某个事件触发，则调用事件处理对象中的事件处理程序来对事件进行处理

观察者模式与Java事件处理

- 分析
  - 事件源(Event Source) ：Subject
    - 例如：JButton，addActionListener()：注册方法，fireXXX()：通知方法
  - 事件监听器(Event Listener)：Observer
    - 例如：ActionListener，actionPerformed()：响应方法
  - 事件处理类(Event Handling Class)：ConcreteObserver
    - 例如：LoginHandling：实现ActionListener接口

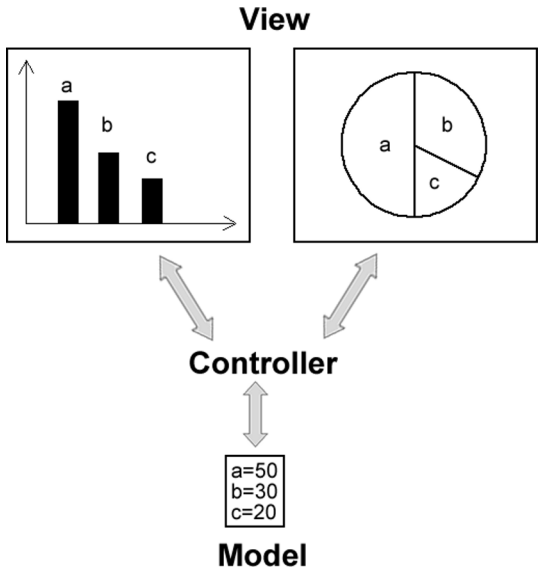
观察者模式与Java事件处理

- 结构



观察者模式与MVC

- MVC(Model-View-Controller)架构
  - 模型(Model)，视图(View)和控制器(Controller)
  - 模型可对应于观察者模式中的观察目标，而视图对应于观察者，控制器可充当两者之间的中介者
  - 当模型层的数据发生改变时，视图层将自动改变其显示内容



MVC结构示意图

观察者模式的优缺点与适用环境

- 模式优点
  - 可以实现表示层和数据逻辑层的分离
  - 在观察目标和观察者之间建立一个抽象的耦合
  - 支持广播通信，简化了一对多系统设计的难度
  - 符合开闭原则，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便

观察者模式的优缺点与适用环境

- 模式缺点
  - 将所有的观察者都通知到会花费很多时间
  - 如果存在循环依赖时可能导致系统崩溃
  - 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，而只是知道观察目标发生了变化

观察者模式的优缺点与适用环境

- 模式适用环境
  - 一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用
  - 一个对象的改变将导致一个或多个其他对象发生改变，且并不知道具体有多少对象将发生改变，也不知道这些对象是谁
  - 需要在系统中创建一个触发链