软件设计模式概述

一个设计模式有四个基本元素

① 模式名

② 问题

③ 解决方案

④ 效果

		目的		
		创建型	结构型	行为型
范围	类	Factory Method 工厂方法	Adapter 适配器（类）	Interpreter 解释器 Template Method 模板方法
	对象	Abstract Factory 抽象工厂 Builder 生成器 Prototype 原型 Singleton 单例	Adapter 适配器（对象） Bridge 桥接 Composite 组合 Decorator 装饰 Facade 外观 Flyweight 享元 Proxy 代理	Chain of Responsibility 职责链 Commands 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

创建型模式

Singleton单例模式

对于系统中的某些类来说，只有一个实例很重要。 它提供全局访问的方法

单例模式的要点有三个：

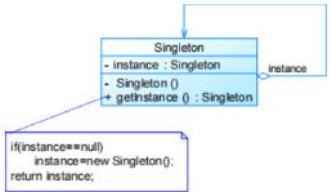
① 某个类只能有一个实例；

② 它必须自行创建这个实例；

③ 它必须自行向整个系统提供这个实例。

私有构造函数

静态私有成员变量与静态公有的工厂方法




```
public class Singleton
{
    private static Singleton instance=null; //静态私有成员变量

    //私有构造函数
    private Singleton()
    {
    }

    //静态公有工厂方法，返回唯一实例
    public static Singleton getInstance()
    {
        if (instance==null)
            instance=new Singleton();
        return instance;
    }
}
```

在现实生活中，居民身份证号码具有唯一性，同一个人不允许有多个身份证号码，第一次申请身份证时将给居民分配一个身份证号码，如果之后因为遗失等原因补办时，还是使用原来的身份证号码，不会产生新的号码。

现使用单例模式模拟该场景。

```
public class IdentityCardNo
{
    private static IdentityCardNo instance=null;
    private String no;

    private IdentityCardNo()
    {
    }

    public static IdentityCardNo getInstance()
    {
        if(instance==null)
        {
            System.out.println("第一次办理身份证，分配新号码！");
            instance=new IdentityCardNo();
            instance.setIdentityCardNo("No:000011112222");
        }
        else
        {
            System.out.println("重复办理身份证，获取旧号码！");
        }
        return instance;
    }
```

```
private void setIdentityCardNo(String no)
{
    this.no=no;
}

public String getIdentityCardNo()
{
    return this.no;
}
```

优点

提供了对唯一实例的受控访问。单例类封装了它的唯一实例，它可以严格控制客户怎样以及何时访问它。

在系统内存中只存在一个对象，可以节约系统资源，对于需要频繁创建和销毁的对象，单例模式可以提高系统性能。

允许可变数目的实例。可以基于单例模式进行扩展，使用与单例控制相似的方法获得指定个数的对象实例。

缺点

单例模式中没有抽象层，因此单例类的扩展有很大的困难。

单例类的职责过重，在一定程度上违背了“单一职责原则”。单例类既充当工厂角色，提供工厂方法，同时又充当产品角色，包含业务方法，将产品的创建和产品的本身的功能融合到一起。

滥用单例将带来一些负面问题。如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；很多面向对象语言的运行环境都提供了自动垃圾回收的技术，如果实例化的对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致对象状态的丢失

适用环境

系统只需要一个实例对象。如系统要求提供一个唯一的序列号生成器，或者需要考虑资源消耗太大而只允许创建一个对象。

客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。

在一个系统中要求一个类只有一个实例时才应当使用单例模式。如果一个类可以有几个实例共存，就需要对单例模式进行改进，使之成为多例模式。

结构型模式

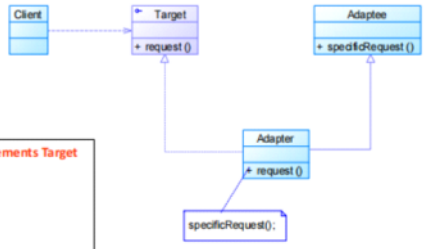
Adaptor适配器模式

客户端可以通过目标类的接口访问它所提供的服务

适配器把客户类的请求转化为对适配者的相应接口的调用。将一个接口转换成客户希望的另一个接口，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。适配器模式是两个不兼容的接口之间的桥梁。

多重继承

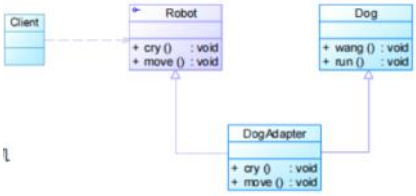
- 类适配器模式实现




```
public class Adapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest();
    }
}
```

现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法cry()、机器人移动方法move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑。

使用类适配器模式进行设计。




```
class Dog{
  public void wang() {
    System.out.println("狗汪汪叫");
  }
  public void run() {
    System.out.println("狗跑");
  }
}
```

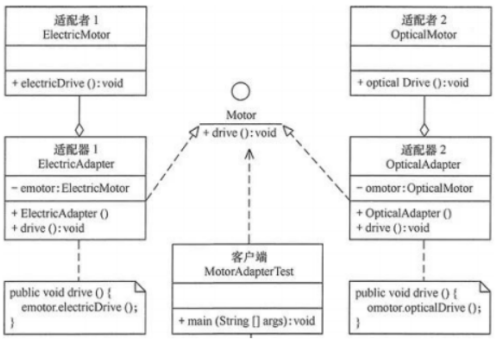
```
public interface Robot {
  public void cry();
  public void move();
}
```

类适配器模式

```
class DogAdapter extends Dog implements Robot{
  public void cry() {
    System.out.println("仿生机器人： ");
    super.wang();
  }
  public void move() {
    System.out.println("仿生机器人： ");
    super.run();
  }
}
```

新能源汽车的发动机有电能发动机（Electric Motor）和光能发动机（Optical Motor）。电能发动机的驱动方法 electricDrive()是用电能驱动，而光能发动机的驱动方法opticalDrive()是用光能驱动。客户端希望用统一的发动机驱动方法drive()访问这两种发动机。

对象适配器模式




```
interface Motor
{
    public void drive();
}
```

```
class ElectricAdapter implements Motor
{
    private ElectricMotor emotor;
    public ElectricAdapter()
    {
        emotor=new ElectricMotor();
    }
    public void drive()
    {
        emotor.electricDrive();
    }
}
```

```
class ElectricMotor
{
    public void electricDrive()
    {
        System.out.println("电能发动机驱动汽车！");
    }
}
```

```
class OpticalAdapter implements Motor
{
    private OpticalMotor omotor;
    public OpticalAdapter()
    {
        omotor=new OpticalMotor();
    }
    public void drive()
    {
        omotor.opticalDrive();
    }
}
```

```
class OpticalMotor
{
    public void opticalDrive()
    {
        System.out.println("光能发动机驱动汽车！");
    }
}
```

如何选择类适配器与对象适配器

判断标准主要有两个：Adaptee接口的个数；Adaptee和Target的契合程度

如果Adaptee接口不多，则两种实现方式都可以；
如果Adaptee接口很多，而且Adaptee和Target接口定义大部分相同，则推荐使用类适配器，因为Adapter复用父类Adaptee的接口，比起对象适配器的实现方式，Adapter的代码量要少一些。

如果Adaptee接口很多，而且Adaptee和Target接口定义大部分不相同，则推荐使用对象适配器，因为组合结构比继承更加灵活

优点

适配器可以使由于接口不兼容而不能交互的类可以一起工作；提高了类的复用；增加了类的透明度；灵活性好

缺点

过多地使用适配器，会让系统非常零乱，不易整体进行把握。
比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。
因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

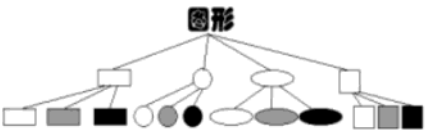
适用环境

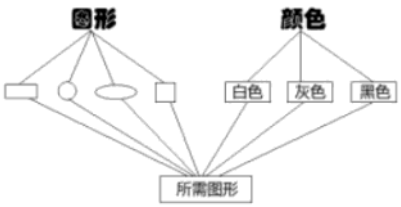
系统需要使用现有的类，而此类的接口不符合系统的需要。
想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些将来不可预见的类协同工作，这些源类不一定有一致的接口。
通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

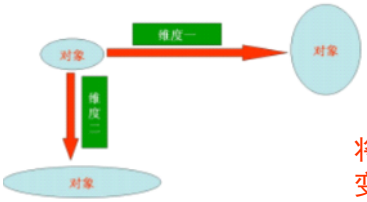
Bridge桥接模式

桥接模式将继承关系转换为关联关系，从而降低了类与类之间的耦合，减少了代码编写量。

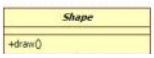
组合或者聚合关系



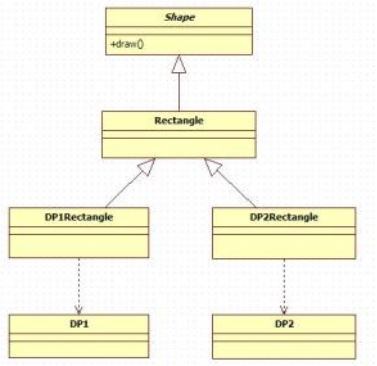


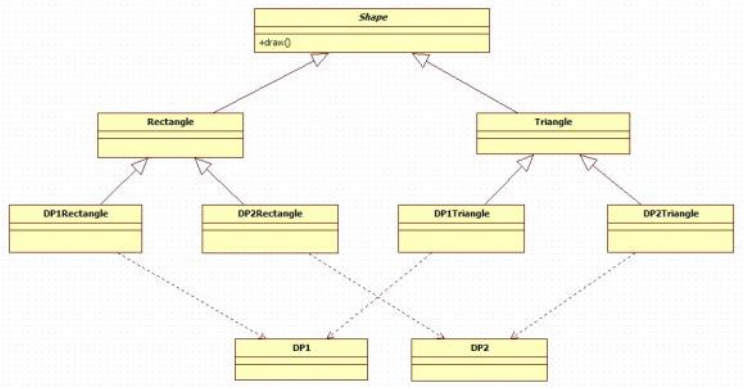


将抽象部分与它的实现部分分离，使它们都可以独立地变化。

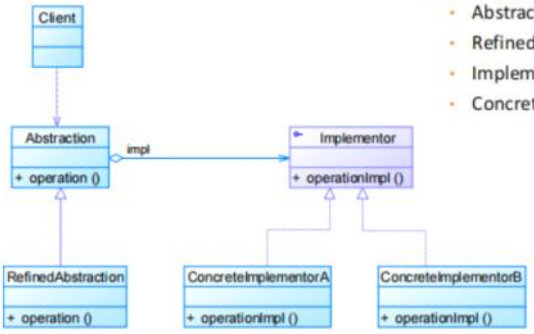


变化。





- 桥接模式包含如下角色：
  - Abstraction：抽象类
  - RefinedAbstraction：扩充抽象类
  - Implementor：实现类接口
  - ConcreteImplementor：具体实现类




```
public interface Implementor
{
    public void operationImpl();
}
```

```
public abstract class Abstraction
{
    protected Implementor impl;

    public void setImpl(Implementor impl)
    {
        this.impl=impl;
    }

    public abstract void operation();
}
```

```
public class RefinedAbstraction extends Abstraction
{
    public void operation()
    {
        //代码
        impl.operationImpl();
        //代码
    }
}
```

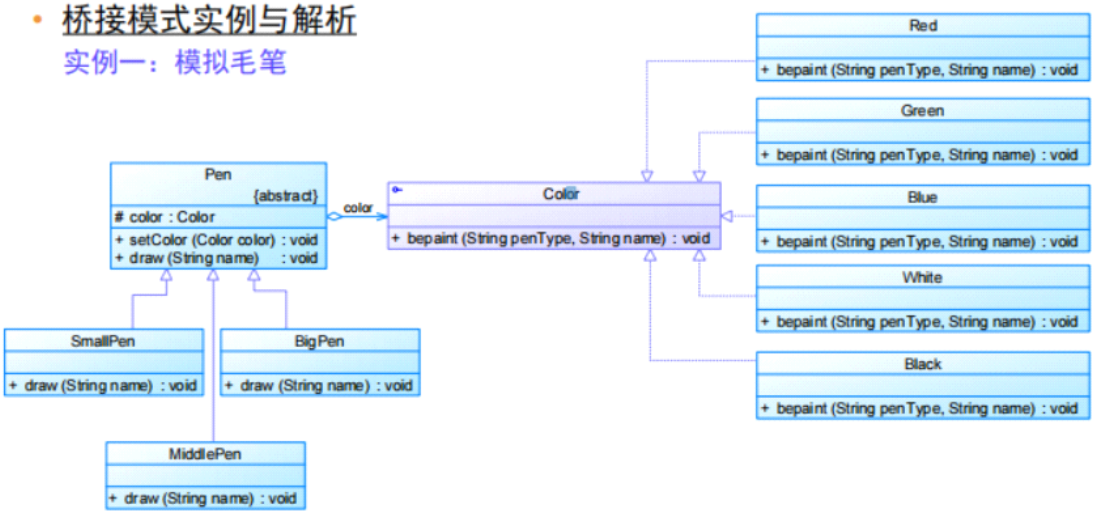
实例一：模拟毛笔

现需要提供大中小3种型号的画笔，能够绘制5种不同颜色，如果使用蜡笔，我们需要准备3*5=15支蜡笔，也就是说必须准备15个具体的蜡笔类。而如果使用毛笔的话，只需要3种型号的毛笔，外加5个颜料盒，用3+5=8个类就可以实现15支蜡笔的功能。

本实例使用桥接模式来模拟毛笔的使用过程。

- 桥接模式实例与解析

实例一：模拟毛笔




```java
public abstract class Pen
{
    protected Color color;
    public void setColor(Color color)
    {
        this.color=color;
    }
    public abstract void draw(String name);
}
```

```java
public class BigPen extends Pen
{
    public void draw(String name)
    {
        String penType="大号毛笔绘制";
        this.color.bepaint(penType,name);
    }
}
```

```java
public interface Color
{
    void bepaint(String penType,String name);
}
```

```java
public class Blue implements Color
{
    public void bepaint(String penType,String name)
    {
        System.out.println(penType + "蓝色的"+ name + ".");
    }
}
```

实例二：跨平台视频播放器

如果需要开发一个跨平台视频播放器，可以在不同操作系统平台（如：Windows、Linux、Unix等）上播放多种格式的视频文件，常见的视频格式包括MPEG、RMVB、AVI、WMV等。

现使用桥接模式设计该播放器。

优点

分离抽象及其实现部分。

桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。

实现细节对客户透明，可以对用户隐藏实现细节

桥接模式是比多继承方案更好的解决方法。多继 承方案违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，而且多继承结构中类的个数非常庞大。

缺点

桥接模式的引入会增加系统的理解与设计难度。由于聚合关系建立在抽象层，要求开发者针对抽象进行设计与编程

桥接模式要求正确识别出系统中两个独立变化的维度，其使用范围具有一定的局限性

模式适用环境

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用

行为型模式

职责链模式

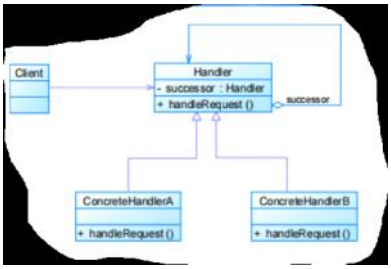
职责链模式可以将请求的处理者组织成一条链

将请求的发送者和请求的处理者解耦

职责链可以是一条直线、一个环或者一个树形结构，最常见的职责链是直线型，即沿着一条单向的链来传递请求

职责链模式包含如下角色：

- Handler：抽象处理者
- ConcreteHandler：具体处理者
- Client：客户类



抽象处理者

```
public abstract class Handler
{
    protected Handler successor;

    public void setSuccessor(Handler successor)
    {
        this.successor=successor;
    }

    public abstract void handleRequest(String request);
}
```

具体处理者

```
public class ConcreteHandler extends Handler
{
    public void handleRequest(String request)
    {
        if (请求request满足条件)
        {
            …… //处理请求；
        }
        else
        {
            this.successor.handleRequest(request);
            //转发请求
        }
    }
}
```

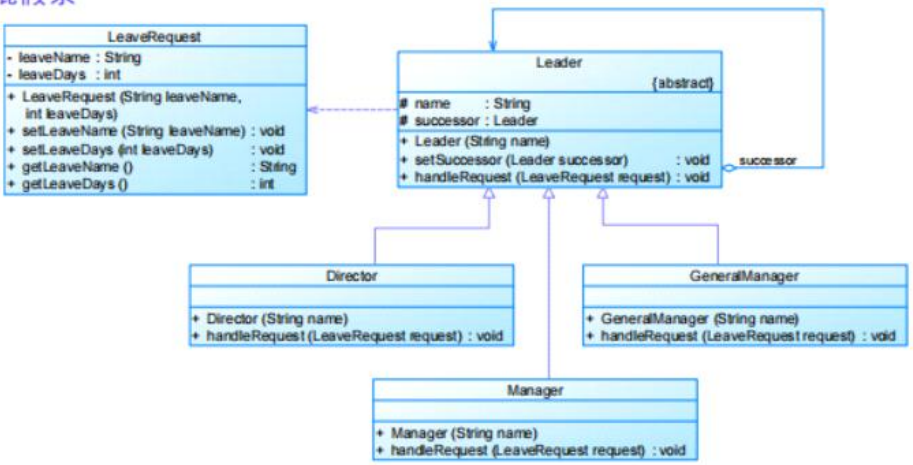
实例：审批假条

某OA系统需要提供一个假条审批的模块：如果员工请假天数小于3天，主任可以审批该假条；如果员工请假天数大于等于3天，小于10天，经理可以审批；如果员工请假天数大于等于10天，小于30天，总经理可以审批；如果超过30天，总经理也不能审批，提示相应的拒绝信息。

现使用职责链模式设计该假条审批模块。

- 职责链模式实例与解析

实例：审批假条




```
public abstract class Leader
{
    protected String name;
    protected Leader successor;
    public Leader(String name)
    {
        this.name=name;
    }
    public void setSuccessor(Leader successor)
    {
        this.successor=successor;
    }
    public abstract void handleRequest(LeaveRequest request);
}
```

```
public class Director extends Leader
{
    public Director(String name)
    {
        super(name);
    }
    public void handleRequest(LeaveRequest request)
    {
        if(request.getLeaveDays()<3)
        {
            System.out.println("主任" + name + "审批员工" + request.getLeaveName());
        }
        else
        {
            if(this.successor!=null)
            {
                this.successor.handleRequest(request);
            }
        }
    }
}
```

```
public class Client
{
    public static void main(String args[])
    {
        Leader objDirector,objManager,objGeneralManager;

        objDirector=new Director("王明");
        objManager=new Manager("赵强");
        objGeneralManager=new GeneralManager("李波");


        objDirector.setSuccessor(objManager);
        objManager.setSuccessor(objGeneralManager);


        LeaveRequest lr1=new LeaveRequest("张三",2);
        objDirector.handleRequest(lr1);

        LeaveRequest lr2=new LeaveRequest("李四",5);
        objDirector.handleRequest(lr2);

        LeaveRequest lr3=new LeaveRequest("王五",15);
        objDirector.handleRequest(lr3);

        LeaveRequest lr4=new LeaveRequest("赵六",25);
        objDirector.handleRequest(lr4);
    }
}
```

```
public class Manager extends Leader
{
    public Manager(String name)
    {
        super(name);
    }
    public void handleRequest(LeaveRequest request)
    {
        if(request.getLeaveDays()<10)
        {
            System.out.println("经理" + name + "审批员工" + request.getLeaveName());
        }
        else
        {
            if(this.successor!=null)
            {
                this.successor.handleRequest(request);
            }
        }
    }
}
```

优点

降低耦合度

增强系统可扩展性，增加新的请求处理类很方便

增强给对象指派职责的灵活性

简化对象的相互连接

缺点

不能保证请求一定被接收。

对比较长的职责链，系统性能将受到一定影响。

模式适用环境

- 有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。
- 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可动态指定一组对象处理请求，或添加新的处理者。

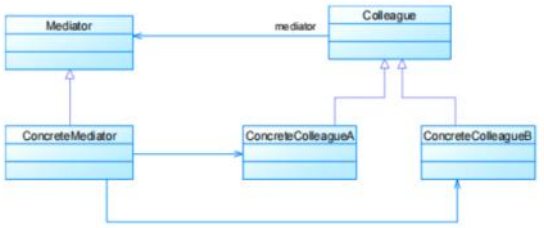
Mediator中介者模式

为了减少对象之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式

中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持。

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。该协调作用属于中介者在行为上的支持。

- Mediator：抽象中介者
- ConcreteMediator：具体中介者
- Colleague：抽象同事类
- ConcreteColleague：具体同事类




```
public abstract class Mediator
{
    protected ArrayList colleagues;
    public void register(Colleague colleague)
    {
        colleagues.add(colleague);
    }
    public abstract void relay();
}
```

```
public class ConcreteMediator extends Mediator
{
    public void relay()
    {
        ......
        colleagues.get(0).method1();
        ......
    }
}
```

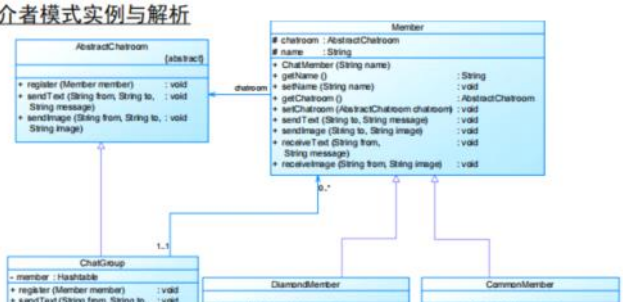
```
public abstract class Colleague
{
        protected Mediator mediator;
        public Colleague(Mediator mediator)
        {
                this.mediator=mediator;
        }
        public abstract void method1(); //自有方法
        public abstract void method2(); //依赖方法
}
```

```
public class ConcreteColleague extends Colleague
{
        public ConcreteColleague(Mediator mediator)
        {
                super(mediator);
        }
        public void method1()   //自有方法
        {
                ......
        }
        public void method2()   //依赖方法
        {
                mediator.relay();
        }
}
```

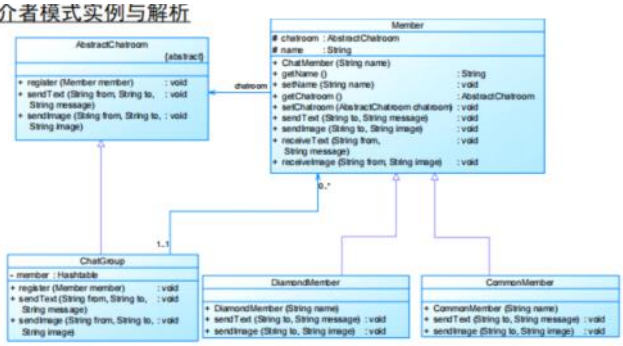
某论坛系统欲增加一个虚拟聊天室，允许论坛会员通过该聊天室进行信息交流，普通会员(CommonMember)可以给其他会员发送文本信息，钻石会员(DiamondMember)既可以给其他会员发送文本信息，还可以发送图片信息。该聊天室可以对不雅字符进行过滤；还可以对发送的图片大小进行控制。

用中介者模式设计该虚拟聊天室。

中介者模式实例与解析



中介者模式实例与解析




```
public abstract class AbstractChatroom
{
    public abstract void register(Member member);
    public abstract void sendText(String from,String to,String message);
    public abstract void sendImage(String from,String to,String message);
}
```

```
public class ChatGroup extends AbstractChatroom
{
    private Hashtable members=new Hashtable();

    public void register(Member member)
    {
        if(!members.contains(member))
        {
            members.put(member.getName(),member);
            member.setChatroom(this);
        }
    }

    public void sendText(String from,String to,String message)
    {
        Member member=(Member)members.get(to);
        String newMessage=message;

        member.receiveText(from,newMessage);
    }

    public void sendImage(String from,String to,String image)
    {
        Member member=(Member)members.get(to);
        //模拟图片太小判断
        if(image.length()>5)
        {
            System.out.println("图片太大，发送失败！");
        }
        else
        {
            member.receiveImage(from,image);
        }
    }
}
```

```
public abstract class Member
{
    protected AbstractChatroom chatroom;
    protected String name;

    public Member(String name)
    {
        this.name=name;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name=name;
    }

    public AbstractChatroom getChatroom()
    {
        return chatroom;
    }

    public void setChatroom(AbstractChatroom chatroom)
    {
        this.chatroom=chatroom;
    }

    public abstract void sendText(String to,String message);
    public abstract void sendImage(String to,String image);

    public void receiveText(String from,String message)
    {
        System.out.println(from + "发送文本给" + this.name + "，内容为：" + message);
    }

    public void receiveImage(String from,String image)
    {
        System.out.println(from + "发送图片给" + this.name + "，内容为：" + image);
    }
}
```

```
public class CommonMember extends Member
{
    public CommonMember(String name)
    {
        super(name);
    }

    public void sendText(String to,String message)
    {
        System.out.println("普通会员发送信息，");
        chatroom.sendText(name,to,message);  //发送
    }

    public void sendImage(String to,String image)
    {
        System.out.println("普通会员不能发送图片！");
    }
}
```

```
public class DiamondMember extends Member
{
    public DiamondMember(String name)
    {
        super(name);
    }

    public void sendText(String to,String message)
    {
        System.out.println("钻石会员发送信息，");
        chatroom.sendText(name,to,message);  //发送
    }

    public void sendImage(String to,String image)
    {
        System.out.println("钻石会员发送图片，");
        chatroom.sendImage(name,to,image);  //发送
    }
}
```

优点

简化了对象之间的交互。
将各个类解耦。
减少子类生成。
可以简化各同事类的设计和实现。

缺点

在具体中介者类中包含了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。

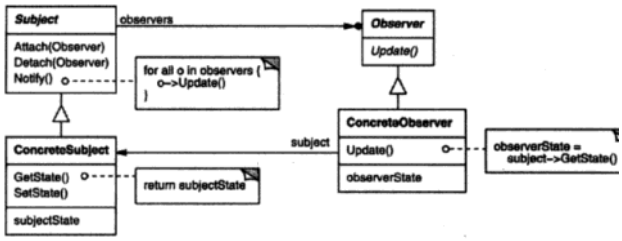
模式适用环境

一组对象以定义良好但复杂的方式进行通信，产生的相互依赖关系结构混乱且难以理解；

一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象；

想定制一个分布在多个类中的行为，又不想生成太多的子类。

观察者模式




```
interface Observer
{
    void update(String state);
}
```

```
class ConcreteObserver implements Observer
{
    public String state;
    public ConcreteObserver(String state)
    {
        this.state = state;
    }
    @Override
    public void update(String state)
    {
        System.out.println("观察者状态更新为"+state);
    }
}
```

```
abstract class Subject
{
    private List<Observer> list = new ArrayList<>();
    public void attach(Observer observer)
    {
        list.add(observer);
    }

    public void detach(Observer observer)
    {
        list.remove(observer);
    }

    public void notifyObservers(String state)
    {
        list.forEach(t->t.update(state));
    }

    public abstract void change(String newState);
}
```

```
class ConcreteSubject extends Subject
{
    private String state;
    public String getstate()
    {
        return state;
    }

    @Override
    public void change(String newState)
    {
        state = newState;
        System.out.println("被观察者状态为:"+newState);
        notifyObservers(newState);
    }
}
```

```
public static void main(String[] args)
{
    Observer observer1 = new ConcreteObserver("111");
    Observer observer2 = new ConcreteObserver("111");
    Observer observer3 = new ConcreteObserver("111");

    Subject subject = new ConcreteSubject();
    subject.attach(observer1);
    subject.attach(observer2);
    subject.attach(observer3);
    subject.change("2222");
}
```

优点

降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
允许独立地改变目标和观察者。

缺点

通知费时：如果有很多观察者，通知需要耗费较多时间。
循环依赖导致崩溃：如果观察者模式与观察目标之间存在循环依赖，观察目标会

导致触发它们之间进行循环调用，可能导致崩溃。
目标上一个看似无害的操作可能会引起一系列对观察者以及依赖于这些观察者的对象的更新。如果依赖准则的定义或维护不当，常常会引起错误的更新，这些错误通常很难捕捉。

模式适用环境

一个抽象模型有两个方面，其中一方依赖于另一方，将二者封装在独立的对象中，以使它们可以独立地改变和复用。
对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象需要改变。