

设计模式

讲师：高淇

设计模式GOF23

- 将设计者的思维融入大家的学习和工作中，更高层次的思考！
- **创建型模式：**
 - 单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式。
- **结构型模式：**
 - 适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
- **行为型模式：**
 - 模版方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、职责链模式、访问者模式。

单例模式

- **核心作用：**
 - 保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。
- **常见应用场景：**
 - Windows的Task Manager (任务管理器)就是很典型的单例模式
 - windows的Recycle Bin (回收站)也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一一个实例。
 - 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，每次new一个对象去读取。
 - 网站的计数器，一般也是采用单例模式实现，否则难以同步。
 - 应用程序的日志应用，一般都用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
 - 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
 - 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。
 - Application 也是单例的典型应用 (Servlet编程中会涉及到)
 - 在Spring中，每个Bean默认就是单例的，这样做的优点是Spring容器可以管理
 - 在servlet编程中，每个Servlet也是单例
 - 在spring MVC框架/struts1框架中，控制器对象也是单例

单例模式

- 单例模式的优点：
 - 由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决
 - 单例模式可以在系统设置全局的访问点，优化环共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理
- 常见的五种单例模式实现方式：
 - 主要：
 - 饿汉式（线程安全，调用效率高。但是，不能延时加载。）
 - 懒汉式（线程安全，调用效率不高。但是，可以延时加载。）
 - 其他：
 - 双重检测锁式（由于JVM底层内部模型原因，偶尔会出问题。不建议使用）
 - 静态内部类式(线程安全，调用效率高。但是，可以延时加载)
 - 枚举单例(线程安全，调用效率高，不能延时加载)

单例模式

- 饿汉式实现（单例对象立即加载）

```
public class SingletonDemo02 {  
    private static /*final*/ SingletonDemo02 s = new SingletonDemo02();  
  
    private SingletonDemo02(){} //私有化构造器  
  
    public static /*synchronized*/ SingletonDemo02 getInstance(){  
        return s;  
    }  
}  
  
public class Client {  
    public static void main(String[] args) {  
        SingletonDemo02 s = SingletonDemo02.getInstance();  
        SingletonDemo02 s2 = SingletonDemo02.getInstance();  
        System.out.println(s==s2); //结果为true  
    }  
}
```

- 饿汉式单例模式代码中，static变量会在类装载时初始化，此时也不会涉及多个线程对象访问该对象的问题。虚拟机保证只会装载一次该类，肯定不会发生并发访问的问题。因此，可以省略synchronized关键字。
- 问题：如果只是加载本类，而不是要调用getInstance()，甚至永远没有调用，则会造成资源浪费！

单例模式

- 懒汉式实现（单例对象延迟加载）

```
public class SingletonDemo01 {  
    private static SingletonDemo01 s;  
  
    private SingletonDemo01(){} //私有化构造器  
  
    public static synchronized SingletonDemo01 getInstance(){  
        if(s==null){  
            s = new SingletonDemo01();  
        }  
        return s;  
    }  
}
```

- 要点：
 - lazy load! 延迟加载， 懒加载！ 真正用的时候才加载！
- 问题：
 - 资源利用率高了。但是，每次调用getInstance()方法都要同步，并发效率较低。

单例模式

- 双重检测锁实现
- 这个模式将同步内容下方到if内部，提高了执行的效率
不必每次获取对象时都进行同步，只有第一次才同步
创建了以后就没必要了。
- 问题：
- 由于编译器优化原因和JVM底层内部模型原因，
偶尔会出问题。不建议使用。

```
public class SingletonDemo03 {  
  
    private static SingletonDemo03 instance = null;  
  
    public static SingletonDemo03 getInstance() {  
        if (instance == null) {  
            SingletonDemo03 sc;  
            synchronized (SingletonDemo03.class) {  
                sc = instance;  
                if (sc == null) {  
                    synchronized (SingletonDemo03.class) {  
                        if(sc == null) {  
                            sc = new SingletonDemo03();  
                        }  
                    }  
                }  
                instance = sc;  
            }  
        }  
        return instance;  
    }  
  
    private SingletonDemo03() {  
    }  
}
```

单例模式

- 静态内部类实现方式(也是一种懒加载方式)

```
public class SingletonDemo04 {  
  
    private static class SingletonClassInstance {  
        private static final SingletonDemo04 instance = new SingletonDemo04();  
    }  
  
    public static SingletonDemo04 getInstance() {  
        return SingletonClassInstance.instance;  
    }  
  
    private SingletonDemo04() {  
    }  
}
```

- 要点：

- 外部类没有static属性，则不会像饿汉式那样立即加载对象。
- 只有真正调用getInstance(),才会加载静态内部类。加载类时是线程 安全的。instance是static final 类型，保证了内存中只有这样一个实例存在，而且只能被赋值一次，从而保证了线程安全性.
- 兼备了并发高效调用和延迟加载的优势！

单例模式

- 问题：
 - 反射可以破解上面几种(不包含枚举式)实现方式！（可以在构造方法中手动抛出异常控制）
 - 反序列化可以破解上面几种((不包含枚举式))实现方式！
 - 可以通过定义readResolve()防止获得不同对象。
 - 反序列化时，如果对象所在类定义了readResolve()，（实际是一种回调），定义返回哪个对象。

```
public class SingletonDemo01 implements Serializable {
    private static SingletonDemo01 s;

    private SingletonDemo01() throws Exception{
        if(s!=null){
            throw new Exception("只能创建一个对象");
        }
        //通过手动抛出异常，避免通过反射创建多个单例对象！
    }
} //私有化构造器

public static synchronized SingletonDemo01 getInstance() throws Exception{
    if(s==null){
        s = new SingletonDemo01();
    }
    return s;
}

//反序列化时，如果对象所在类定义了readResolve()，（实际是一种回调），定义返回哪个对象。
private Object readResolve() throws ObjectStreamException {
    return s;
}
```

单例模式

- 使用枚举实现单例模式

```
public enum SingletonDemo05 {  
    /**  
     * 定义一个枚举的元素，它就代表了Singleton的一个实例。  
     */  
    INSTANCE;  
    /**  
     * 单例可以有自己的操作  
     */  
    public void singletonOperation(){  
        //功能处理  
    }  
}  
  
public static void main(String[] args) {  
    SingletonDemo05 sd = SingletonDemo05.INSTANCE;  
    SingletonDemo05 sd2 = SingletonDemo05.INSTANCE;  
    System.out.println(sd==sd2);  
}
```

- 优点：
 - 实现简单
 - 枚举本身就是单例模式。由JVM从根本上提供保障！避免通过反射和反序列化的漏洞！
- 缺点：
 - 无延迟加载

单例模式

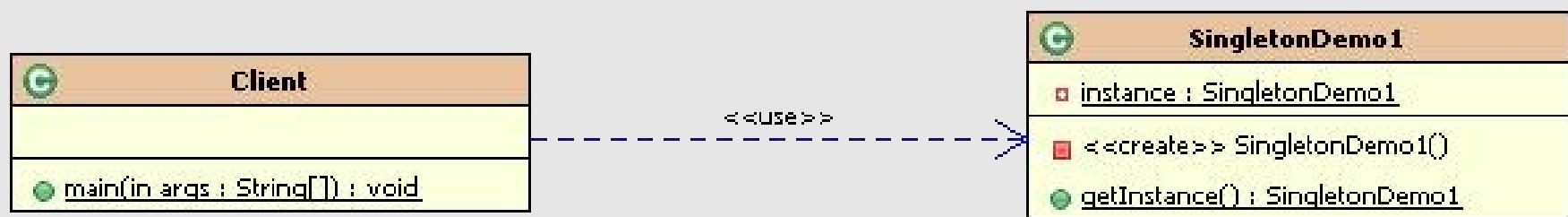
- 常见的五种单例模式在多线程环境下的效率测试
 - 大家只要关注相对值即可。在不同的环境下不同的程序测得值完全不一样

饿汉式	22ms
懒汉式	636ms
静态内部类式	28ms
枚举式	32ms
双重检查锁式	65ms

- CountDownLatch
 - 同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
 - countDown() 当前线程调此方法，则计数减一(建议放在 finally里执行)
 - await()， 调用此方法会一直阻塞当前线程，直到计时器的值为0

单例模式

- 使用myeclipse的UML插件画出类图
 - 大家也可以使用：rational rose 、 metamill等。



单例模式

- 常见的五种单例模式实现方式
 - 主要：
 - 饿汉式（线程安全，调用效率高。但是，不能延时加载。）
 - 懒汉式（线程安全，调用效率不高。但是，可以延时加载。）
 - 其他：
 - 双重检测锁式（由于JVM底层内部模型原因，偶尔会出问题。**不建议使用**）
 - 静态内部类式(线程安全，调用效率高。但是，可以延时加载)
 - 枚举式(线程安全，调用效率高，不能延时加载。并且可以天然的防止反射和反序列化漏洞！)
- 如何选用？
 - 单例对象 占用 资源 少，不需要 延时加载：
 - 枚举式 好于 饿汉式
 - 单例对象 占用 资源 大，需要 延时加载：
 - 静态内部类式 好于 懒汉式

工厂模式

- 工厂模式：
 - 实现了创建者和调用者的分离。
 - 详细分类：
 - 简单工厂模式
 - 工厂方法模式
 - 抽象工厂模式
- 面向对象设计的基本原则：
 - OCP (开闭原则，Open-Closed Principle)：一个软件的实体应当对扩展开放，对修改关闭。
 - DIP (依赖倒转原则，Dependence Inversion Principle)：要针对接口编程，不要针对实现编程。
 - LoD (迪米特法则，Law of Demeter)：只与你直接的朋友通信，而避免和陌生人通信。

工厂模式

- 核心本质：
 - 实例化对象，用工厂方法代替new操作。
 - 将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。
- 工厂模式：
 - 简单工厂模式
 - 用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
 - 工厂方法模式
 - 用来生产同一等级结构中的固定产品。（支持增加任意产品）
 - 抽象工厂模式
 - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）

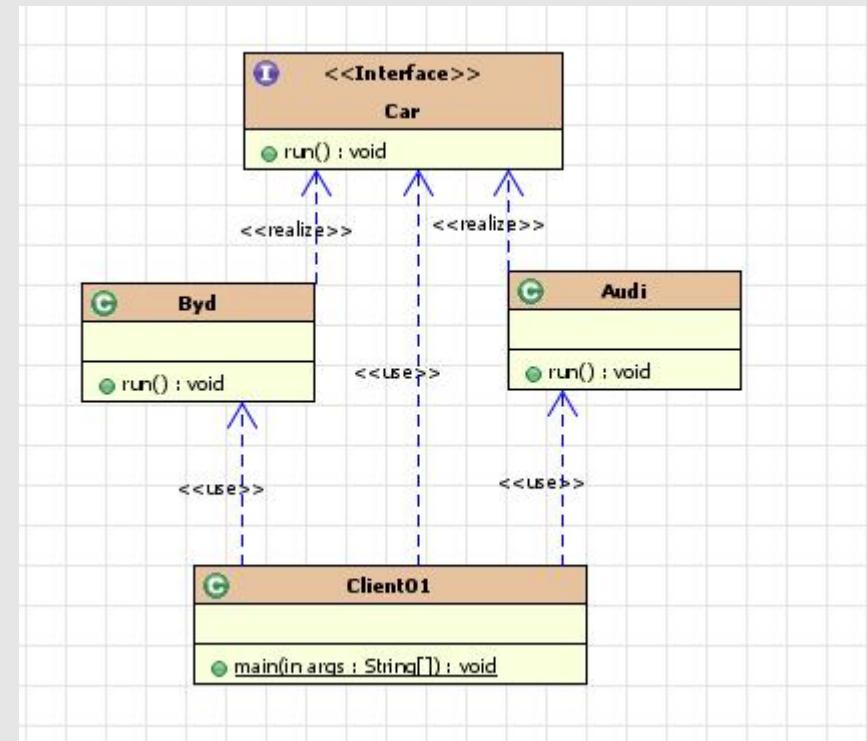
工厂模式

- 不使用简单工厂的情况

```
public class Client01 {    //调用者

    public static void main(String[] args) {
        Car c1 = new Audi();
        Car c2 = new Byd();

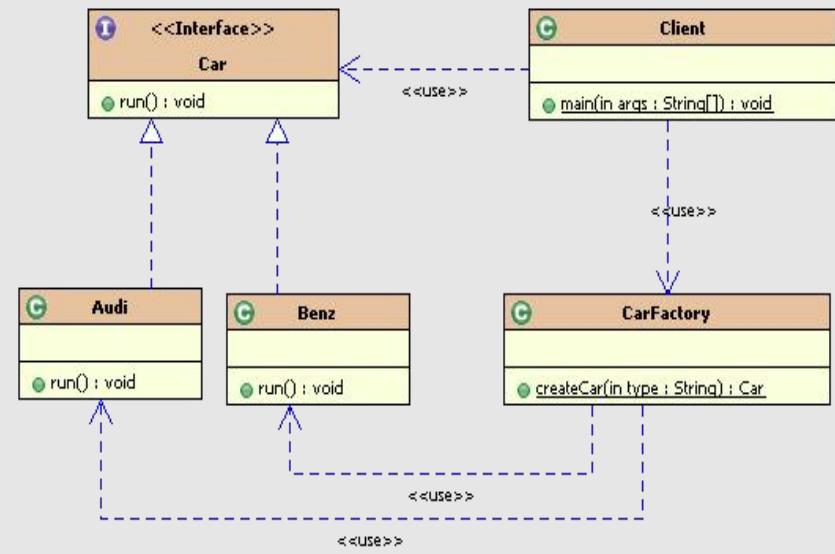
        c1.run();
        c2.run();
    }
}
```



简单工厂模式

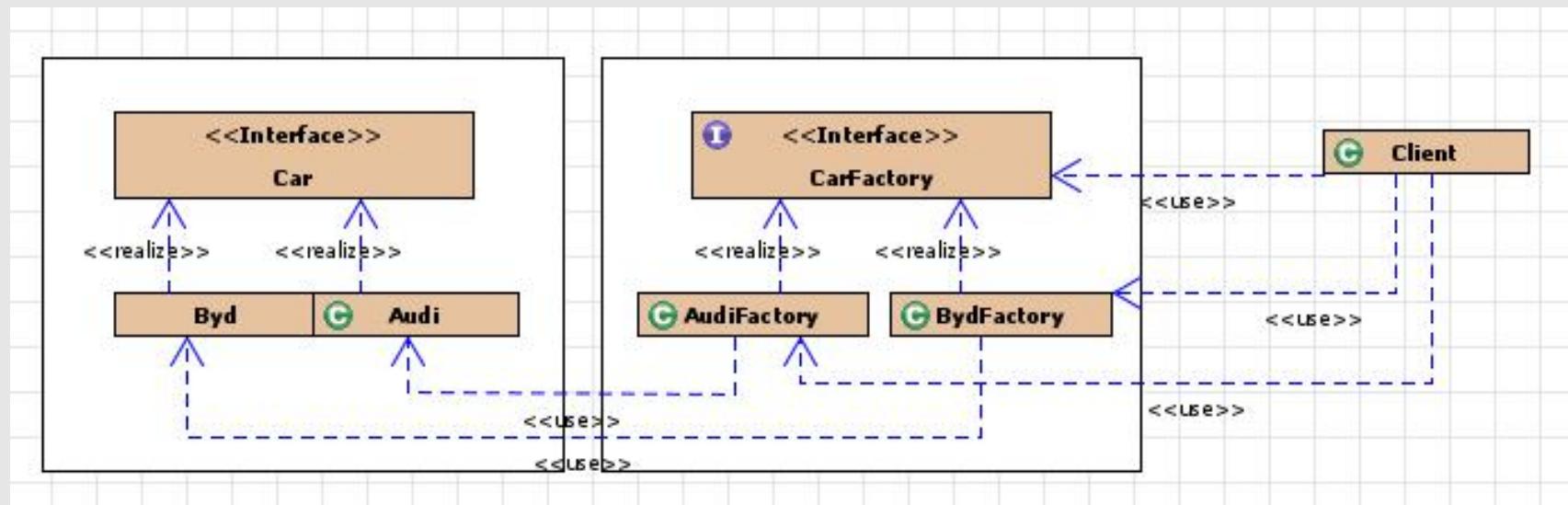
- 要点：
 - 简单工厂模式也叫静态工厂模式，就是工厂类一般是使用静态方法，通过接收的参数的不同来返回不同的对象实例。
 - 对于增加新产品无能为力！不修改代码的话，是无法扩展的。

```
package com.bjsxt.simpleFactory;
public class CarFactory {
    public static Car createCar(String type){
        Car c = null;
        if("奥迪".equals(type)){
            c = new Audi();
        }else if("奔驰".equals(type)){
            c = new Benz();
        }
        return c;
    }
}
public class CarFactory {
    public static Car createAudi(){
        return new Audi();
    }
    public static Car createBenz(){
        return new Benz();
    }
}
```



工厂方法模式

- 工厂方法模式要点：
 - 为了避免简单工厂模式的缺点，不完全满足OCP。
 - 工厂方法模式和简单工厂模式最大的不同在于，简单工厂模式只有一个（对于一个项目或者一个独立模块而言）工厂类，而工厂方法模式有一组实现了相同接口的工厂类。



工厂模式

- 简单工厂模式和工厂方法模式PK:

- **结构复杂度**

从这个角度比较，显然简单工厂模式要占优。简单工厂模式只需一个工厂类，而工厂方法模式的工厂类随着产品类个数增加而增加，这无疑会使类的个数越来越多，从而增加了结构的复杂程度。

- **代码复杂度**

代码复杂度和结构复杂度是一对矛盾，既然简单工厂模式在结构方面相对简洁，那么它在代码方面肯定是比工厂方法模式复杂的了。简单工厂模式的工厂类随着产品类的增加需要增加很多方法（或代码），而工厂方法模式每个具体工厂类只完成单一任务，代码简洁。

- **客户端编程难度**

工厂方法模式虽然在工厂类结构中引入了接口从而满足了OCP，但是在客户端编码中需要对工厂类进行实例化。而简单工厂模式的工厂类是个静态类，在客户端无需实例化，这无疑是个吸引人的优点。

- **管理上的难度**

这是个关键的问题。

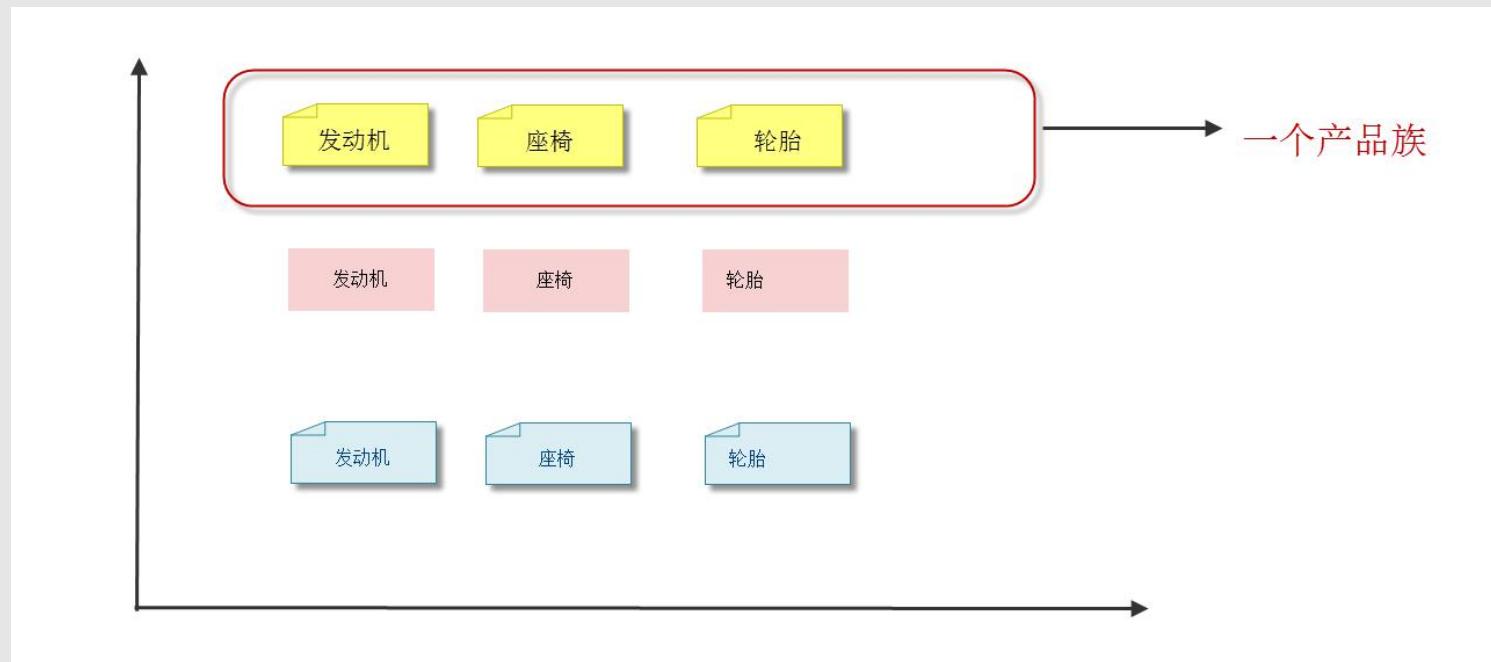
我们先谈扩展。众所周知，工厂方法模式完全满足OCP，即它有非常良好的扩展性。那是否就说明了简单工厂模式就没有扩展性呢？答案是否定的。简单工厂模式同样具备良好的扩展性——扩展的时候仅需要修改少量的代码（修改工厂类的代码）就可以满足扩展性的要求了。尽管这没有完全满足OCP，但我们不需要太拘泥于设计理论，要知道，sun提供的java官方工具包中也有想到多没有满足OCP的例子啊。

然后我们从维护性的角度分析下。假如某个具体产品类需要进行一定的修改，很可能需要修改对应的工厂类。当同时需要修改多个产品类的时候，对工厂类的修改会变得相当麻烦（对号入座已经是个问题了）。反而简单工厂没有这些麻烦，当多个产品类需要修改时，简单工厂模式仍然仅仅需要修改唯一的工厂类（无论怎样都能改到满足要求吧？大不了把这个类重写）。

- 根据设计理论建议：工厂方法模式。但实际上，我们一般都用简单工厂模式。

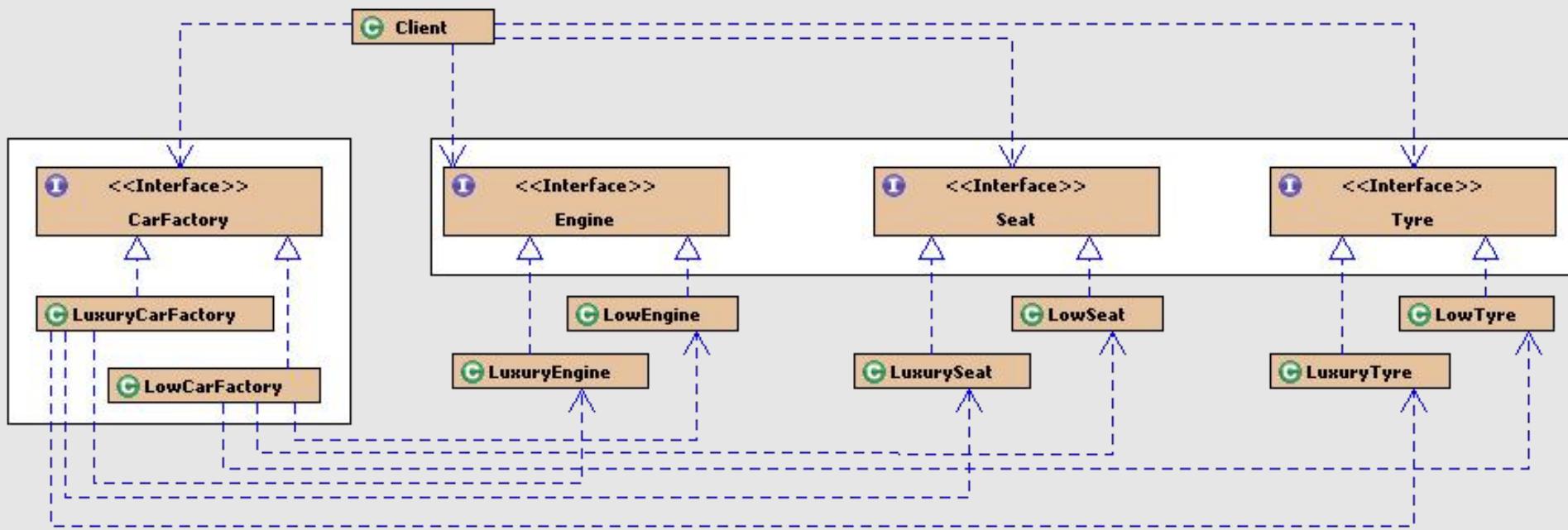
抽象工厂模式

- 抽象工厂模式
 - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）
 - 抽象工厂模式是工厂方法模式的升级版本，在有多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方式。



抽象工厂模式

- 类图



工厂模式

- 工厂模式要点：
 - 简单工厂模式(静态工厂模式)
 - 虽然某种程度不符合设计原则，但实际使用最多。
 - 工厂方法模式
 - 不修改已有类的前提下，通过增加新的工厂类实现扩展。
 - 抽象工厂模式
 - 不可以增加产品，可以增加产品族！
- 应用场景
 - JDK中Calendar的getInstance方法
 - JDBC中Connection对象的获取
 - Hibernate中SessionFactory创建Session
 - spring中IOC容器创建管理bean对象
 - XML解析时的DocumentBuilderFactory创建解析器对象
 - 反射中Class对象的newInstance()

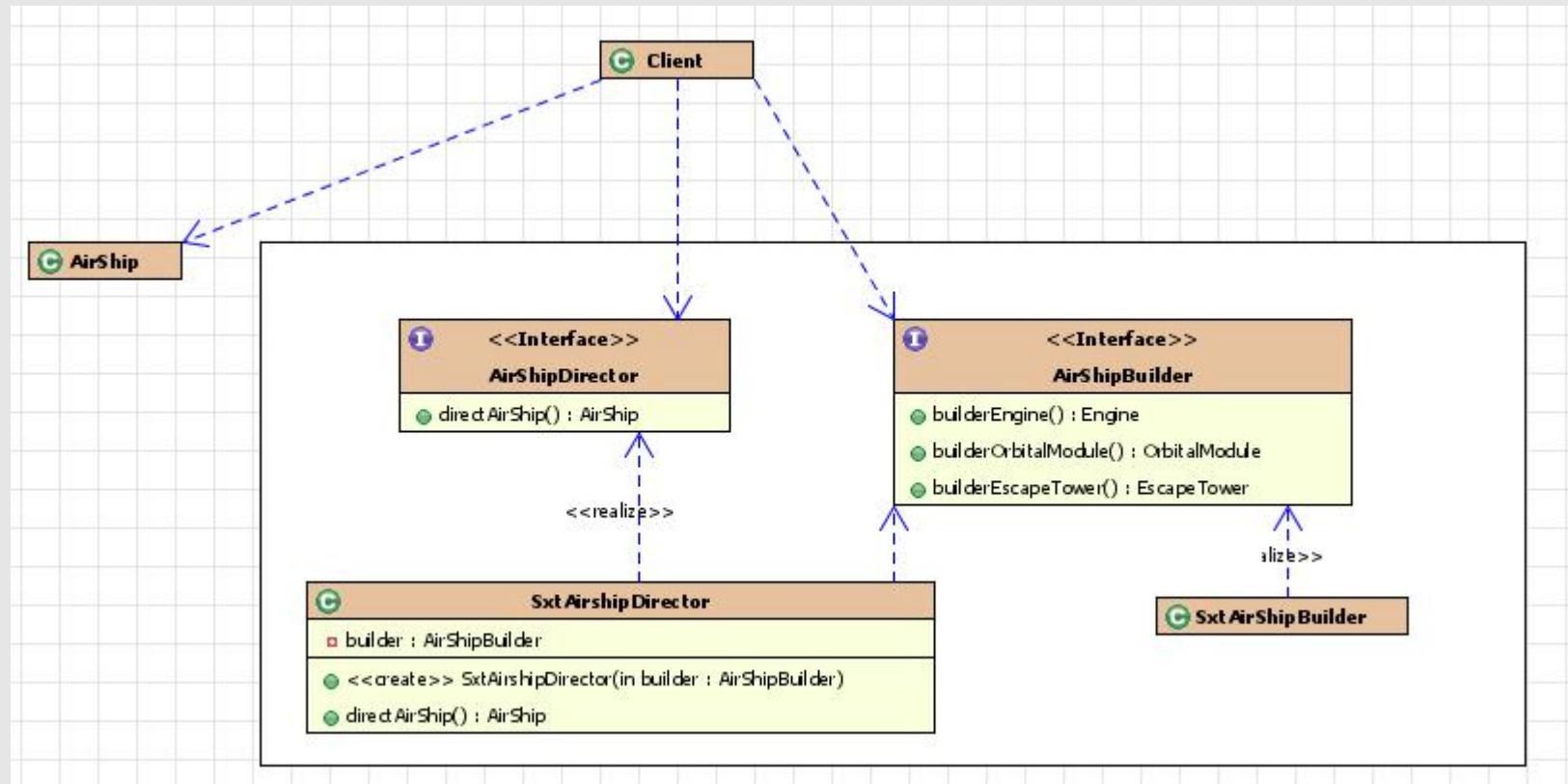
建造者模式

- 场景：
 - 我们要建造一个复杂的产品。比如：神州飞船,Iphone。这个复杂的产品的创建。有这样一个问题需要处理：
 - 装配这些子组件是不是有个步骤问题？
 - 实际开发中，我们所需要的对象构建时，也非常复杂，有很多步骤需要处理时。
- 建造模式的本质：
 - 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。从而可以构造出复杂的对象。这个模式适用于：某个对象的构建过程复杂的情况下使用。
 - 由于实现了构建和装配的解耦。不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用。



建造者模式

- 构建“尚学堂牌”神舟飞船的示例



建造者模式

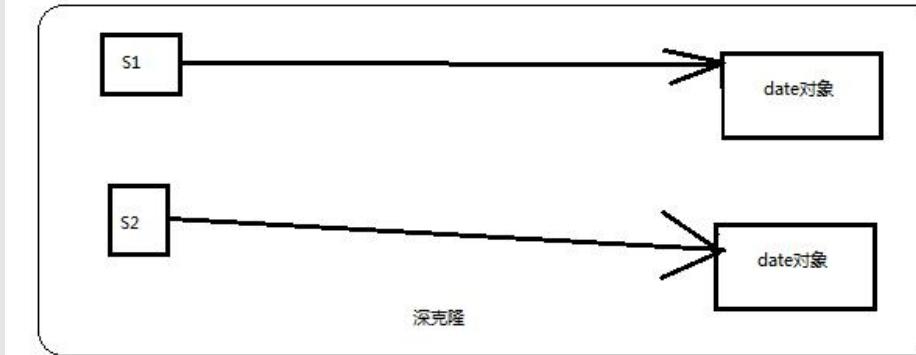
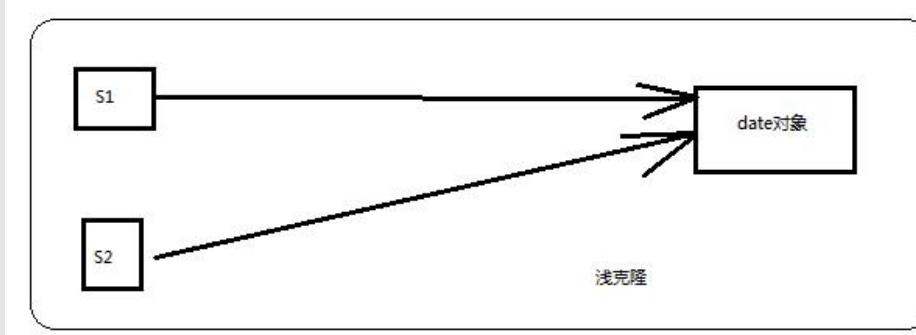
- 开发中应用场景：
 - StringBuilder类的append方法
 - SQL中的PreparedStatement
 - JDOM中，DomBuilder、SAXBuilder

原型模式prototype

- 场景：
 - 思考一下：克隆技术是怎么样的过程？克隆羊多利大家还记得吗？
 - javascript语言中的，继承怎么实现？那里面也有prototype，大家还记得吗？
- 原型模式：
 - 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。
 - 就是java中的克隆技术，以某个对象为原型，复制出新的对象。显然，新的对象具备原型对象的特点
 - 优势有：效率高(直接克隆，避免了重新执行构造过程步骤)。
 - 克隆类似于new，但是不同于new。new创建新的对象属性采用的是默认值。克隆出的对象的属性值完全和原型对象相同。并且克隆出的新对象改变不会影响原型对象。然后，再修改克隆对象的值。
- 原型模式实现：
 - Cloneable接口和clone方法
 - Prototype模式中实现起来最困难的地方就是内存复制操作，所幸在Java中提供了clone()方法替我们做了绝大部分事情。
- 注意用词：克隆和拷贝一回事！

原型模式prototype

- 浅克隆存在的问题
 - 被复制的对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。
- 深克隆如何实现?
 - 深克隆把引用的变量指向复制过的新对象，而不是原有的被引用的对象。
 - 深克隆：让已实现Clonable接口的类中的属性也实现Clonable接口
 - 基本数据类型和String能够自动实现深度克隆（值的复制）



原型模式prototype

- 利用序列化和反序列化技术实现深克隆！

```
Date date = new Date(12312321331L);
Sheep s1 = new Sheep("少利",date);
System.out.println(s1);
System.out.println(s1.getSname());
System.out.println(s1.getBirthday());

//使用序列化和反序列化实现深复制
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(s1);
byte[] bytes = bos.toByteArray();

ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
ObjectInputStream ois = new ObjectInputStream(bis);

Sheep s2 = (Sheep) ois.readObject(); //克隆好的对象!

System.out.println("修改原型对象的属性值");
date.setTime(23432432423L);

System.out.println(s1.getBirthday());

s2.setSname("多利");
System.out.println(s2);
System.out.println(s2.getSname());
System.out.println(s2.getBirthday());
```

原型模式prototype

- 短时间大量创建对象时，原型模式和普通new方式效率测试
- 开发中的应用场景
 - 原型模式很少单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。
 - spring中bean的创建实际就是两种：单例模式和原型模式。（当然，原型模式需要和工厂模式搭配起来）

创建型模式的总结

- **创建型模式：都是用来帮助我们创建对象的！**
 - 单例模式
 - 保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。
 - 工厂模式
 - 简单工厂模式
 - 用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
 - 工厂方法模式
 - 用来生产同一等级结构中的固定产品。（支持增加任意产品）
 - 抽象工厂模式
 - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）
 - 建造者模式
 - 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。从而可以构造出复杂的对象。
 - 原型模式
 - 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式

结构型模式

- **结构型模式：**
 - 核心作用：是从程序的结构上实现松耦合，从而可以扩大整体的类结构，用来解决更大的问题。
 - 分类：
 - 适配器模式、代理模式、桥接模式、装饰模式、组合模式、外观模式、享元模式

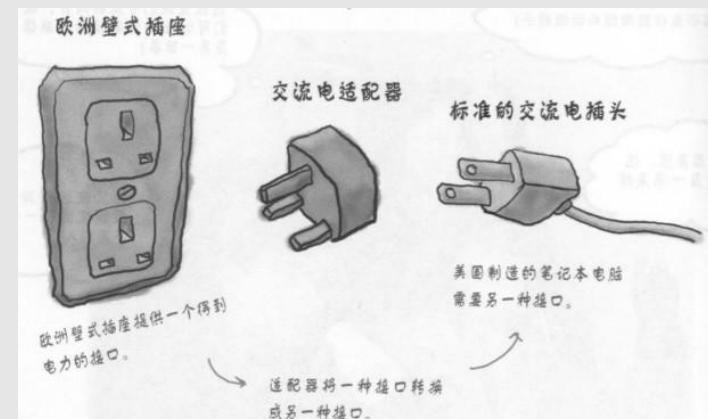
结构型模式总结

• 结构型模式汇总

代理模式	为真实对象提供一个代理，从而控制对真实对象的访问
适配模式	使原本由于接口不兼容不能一起工作的类可以一起工作
桥接模式	处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各个维度可以独立的扩展在抽象层建立关联。
组合模式	将对象组合成树状结构以表示”部分和整体”层次结构，使得客户可以统一的调用叶子对象和容器对象
装饰模式	动态地给一个对象添加额外的功能，比继承灵活
外观模式	为子系统提供统一的调用接口，使得子系统更加容易使用
享元模式	运用共享技术有效的实现管理大量细粒度对象，节省内存，提高效率

适配器adapter模式

- 生活中的场景



适配器adapter模式

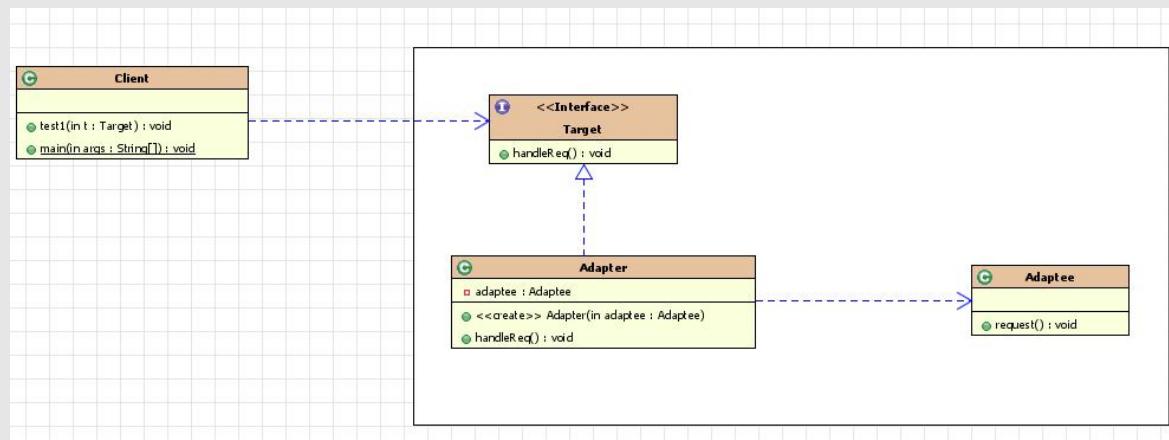
- 什么是适配器模式?
 - 将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。
- 模式中的角色
 - 目标接口 (Target) : 客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口。
 - 需要适配的类 (Adaptee) : 需要适配的类或适配者类。
 - 适配器 (Adapter) : 通过包装一个需要适配的对象，把原接口转换成目标接口。

适配器adapter模式

- 下面的场景，如何解决？



- 适配器模式解决方案：



适配器adapter模式

- 类适配器

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

- 对象适配器

```
class Adapter implements Target{  
    private Adaptee adaptee;  
  
    public Adapter (Adaptee adaptee) {  
        this.adaptee = adaptee;  
    }  
  
    public void request() {  
        this.adaptee.specificRequest();  
    }  
}
```

适配器adapter模式

- 工作中的场景
 - 经常用来做旧系统改造和升级
 - 如果我们的系统开发之后再也不需要维护，那么很多模式都是没必要的，但是不幸的是，事实却是维护一个系统的代价往往是开发一个系统的数倍。
- 我们学习中见过的场景
 - `java.io.InputStreamReader(InputStream)`
 - `java.io.OutputStreamWriter(OutputStream)`

代理模式

- **代理模式(Proxy pattern) :**

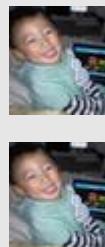
- 核心作用：

- 通过代理，控制对对象的访问！

可以详细控制访问某个（某类）对象的方法，在调用这个方法前做前置处理，调用这个方法后做后置处理。（即：AOP的微观实现！）

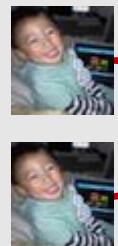
- AOP(Aspect Oriented Programming面向切面编程)的核心实现机制！

从而实现将统一
流程代码放到代
理类中处理



1. 面谈
2. 合同起草
3. 签字，收预付款
4. 安排机票和车辆
5. 唱歌
6. 收尾款

唱歌



1. 面谈
2. 合同起草
3. 签字，收预付款
4. 安排机票和车辆
5. 安排唱歌
6. 收尾款



代理模式

- **代理模式(Proxy pattern) :**

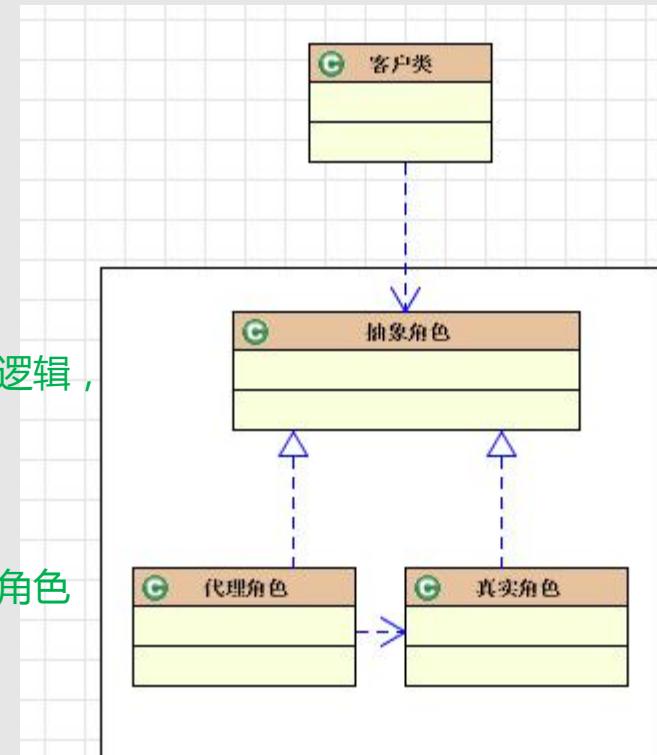
- **核心角色 :**

- **抽象角色**
 - 定义代理角色和真实角色的公共对外方法

- **真实角色**
 - 实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。
 - **关注真正的业务逻辑！**

- **代理角色**
 - 实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

- **将统一的流程控制放到代理角色中处理！**

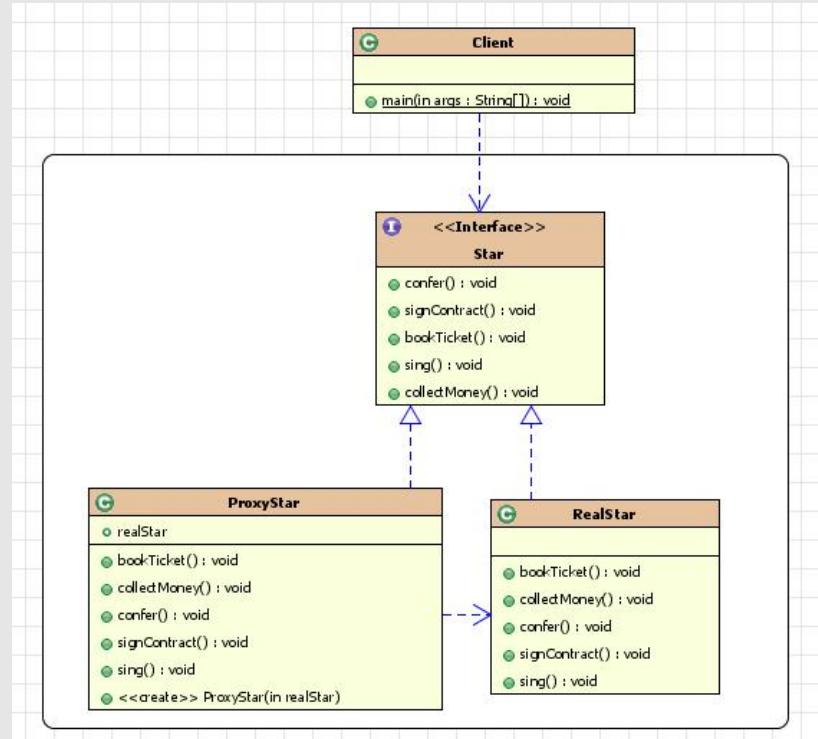


代理模式

- 应用场景：
 - 安全代理：屏蔽对真实角色的直接访问。
 - 远程代理：通过代理类处理远程方法调用(RMI)
 - 延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象。
 - 比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有100MB，在打开文件时不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开。
- 分类：
 - 静态代理(静态定义代理类)
 - 动态代理(动态生成代理类)
 - JDK自带的动态代理
 - javaassist字节码操作库实现
 - CGLIB
 - ASM(底层使用指令，可维护性较差)

静态代理 (static proxy)

- 静态代理(静态定义代理类)



动态代理(dynamic proxy)

- 动态代理(动态生成代理类)
 - JDK自带的动态代理
 - javaassist字节码操作库实现
 - CGLIB
 - ASM(底层使用指令，可维护性较差)

动态代理的优点

- 动态代理相比于静态代理的优点
 - 抽象角色中(接口)声明的所以方法都被转移到调用处理器一个集中的方法中处理，这样，我们可以更加灵活和统一的处理众多的方法。

动态代理(JDK自带的实现)

- JDK自带的动态代理
 - `java.lang.reflect.Proxy`
 - 作用：动态生成代理类和对象
 - `java.lang.reflect.InvocationHandler`(处理器接口)
 - 可以通过`invoke`方法实现对真实角色的代理访问。
 - 每次通过`Proxy`生成代理类对象对象时都要指定对应的处理器对象

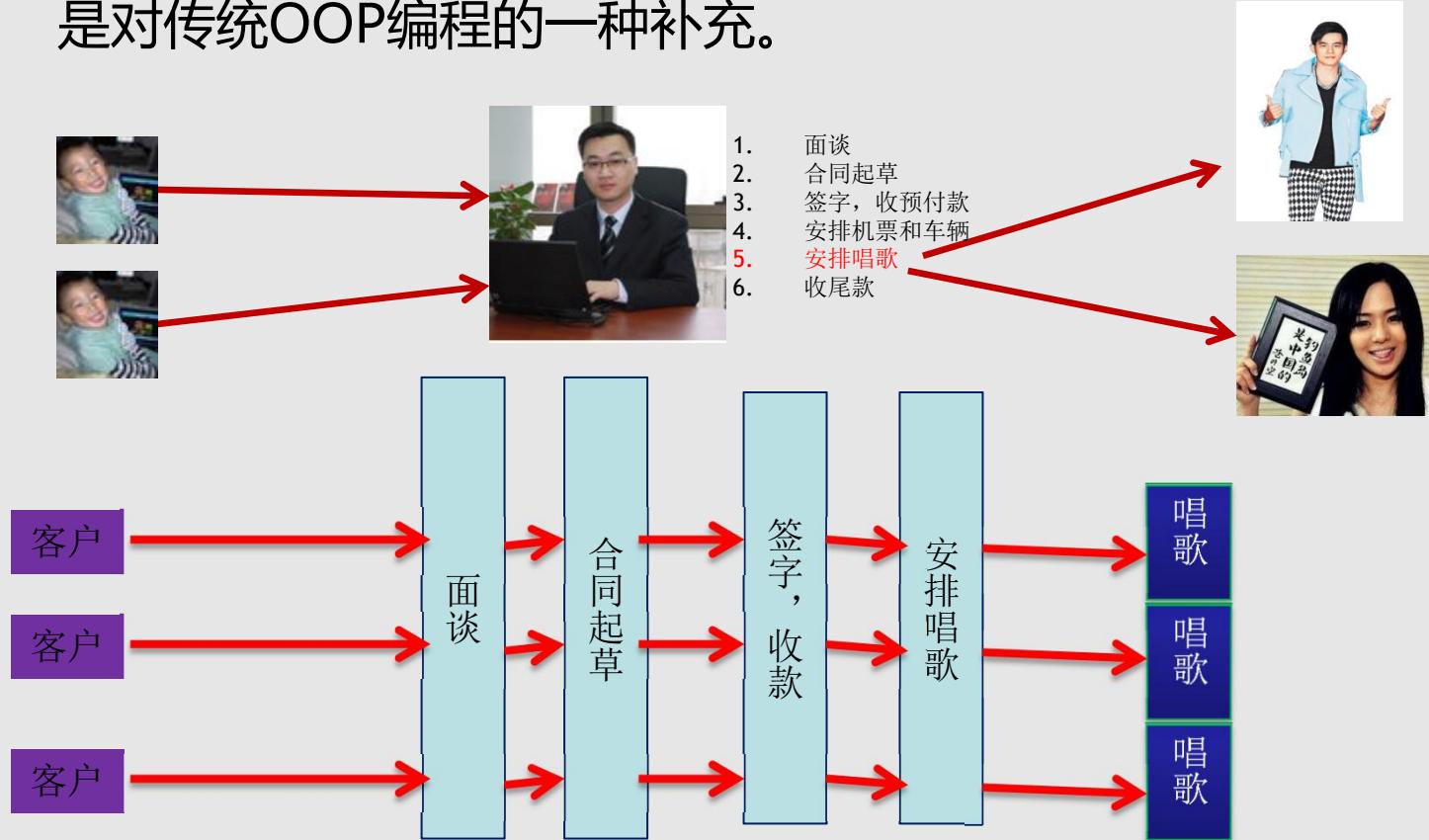
```
Star realStar = new RealStar();
StarHandler handler = new StarHandler(realStar);
Star proxy = (Star) Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),new
Class[]{Star.class},handler);
proxy.sing();
```

代理模式

- 开发框架中应用场景：
 - struts2中拦截器的实现
 - 数据库连接池关闭处理
 - Hibernate中延时加载的实现
 - mybatis中实现拦截器插件
 - AspectJ的实现
 - spring中AOP的实现
 - 日志拦截
 - 声明式事务处理
 - web service
 - RMI远程方法调用
 - ...
- 实际上，随便选择一个技术框架都会用到代理模式！！

面向切面编程AOP介绍

- AOP (Aspect-Oriented Programming , 面向切面的编程)
 - 它是可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。它是一种新的方法论，它是对传统OOP编程的一种补充。

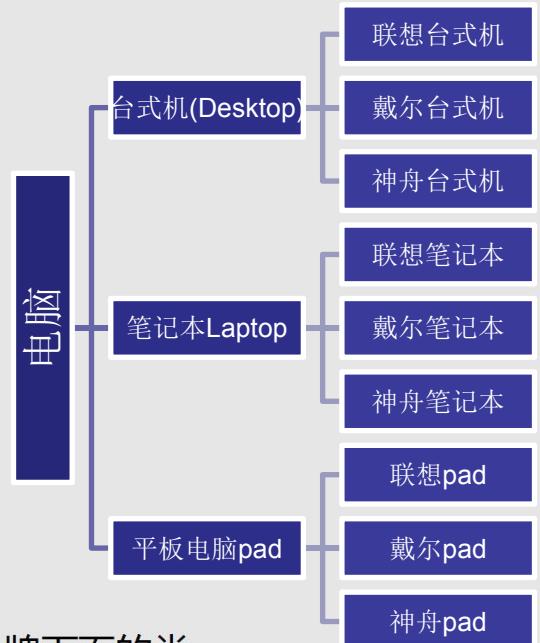


面向切面编程AOP介绍

- 常用术语：
 - 切面 (Aspect) : 其实就是共有功能的实现。
 - 通知 (Advice) : 是切面的具体实现。
 - 连接点 (Joinpoint) : 就是程序在运行过程中能够插入切面的地点。
 - 切入点 (Pointcut) : 用于定义通知应该切入到哪些连接点上。
 - 目标对象 (Target) : 就是那些即将切入切面的对象，也就是那些被通知的对象。
 - 代理对象 (Proxy) : 将通知应用到目标对象之后被动态创建的对象。
 - 织入 (Weaving) : 将切面应用到目标对象从而创建一个新的代理对象的过程。
- 开源的AOP框架
 - AspectJ

桥接模式(bridge)

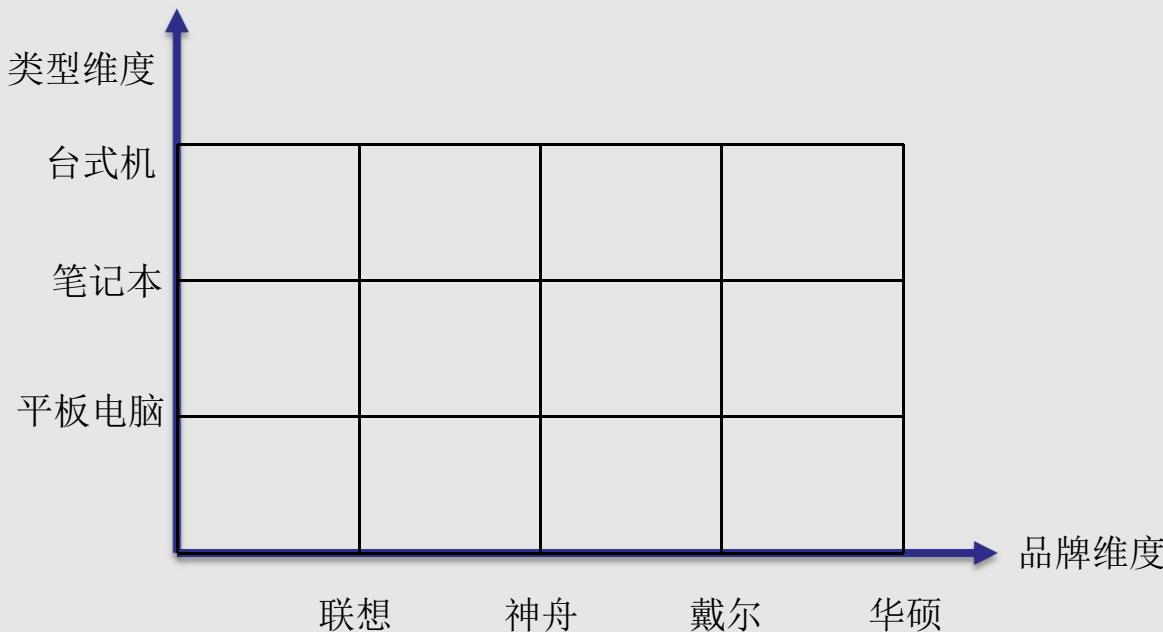
- 场景
 - 商城系统中常见的商品分类，以电脑为类，如何良好的处理商品分类销售的问题？
- 我们可以用**多层继承结构**实现右图的关系。
- 问题：
 - 扩展性问题(类个数膨胀问题)：
 - 如果要增加一个新的电脑类型:智能手机，则要增加各个品牌下面的类。
 - 如果要增加一个新的品牌，也要增加各种电脑类型的类。
 - 违反单一职责原则：
 - 一个类：联想笔记本，有两个引起这个类变化的原因



桥接模式

- 场景分析

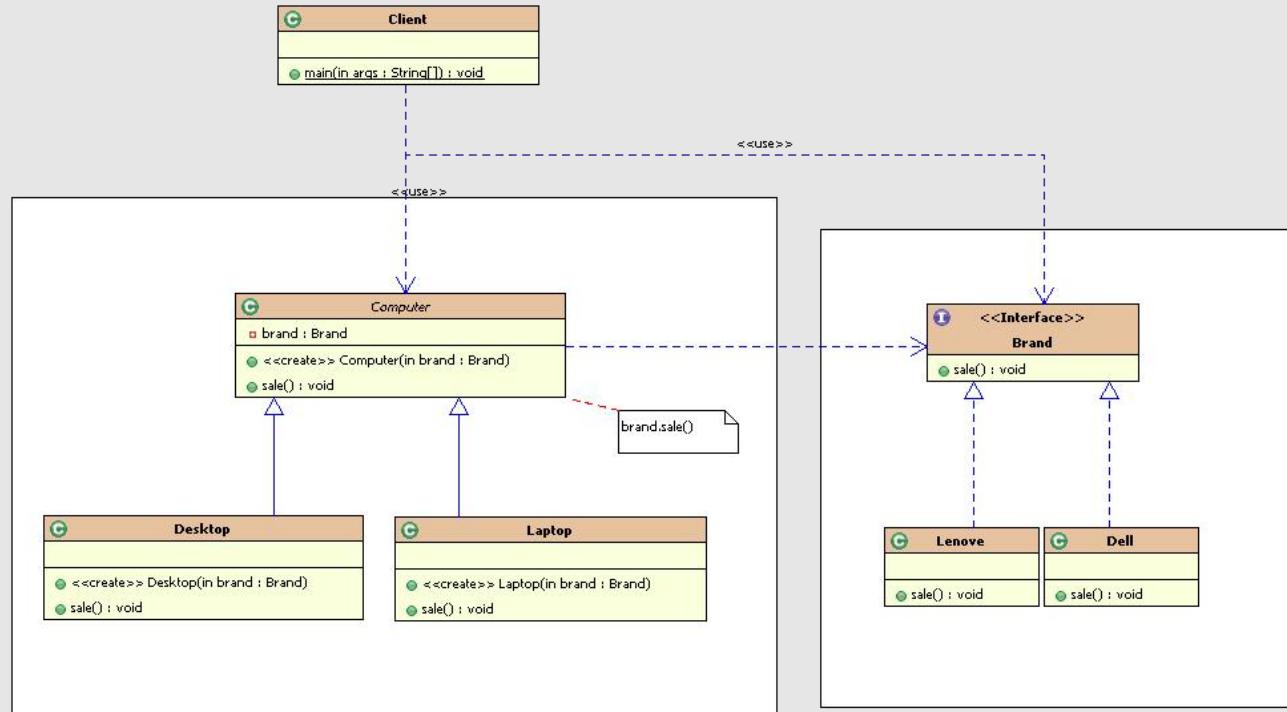
- 商城系统中常见的商品分类，以电脑为类，如何良好的处理商品分类销售的问题？
- 这个场景中有两个变化的维度：电脑类型、电脑品牌。



桥接模式

- 桥接模式核心要点：

- 处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各个维度可以独立的扩展在抽象层建立关联。



桥接模式

- 桥接模式总结：

- 桥接模式可以取代多层继承的方案。多层继承违背了单一职责原则，复用性较差，类的个数也非常多。桥接模式可以极大的减少子类的个数，从而降低管理和维护的成本。
- 桥接模式极大的提高了系统可扩展性，在两个变化维度中任意扩展一个维度，都不需要修改原有的系统，符合开闭原则。



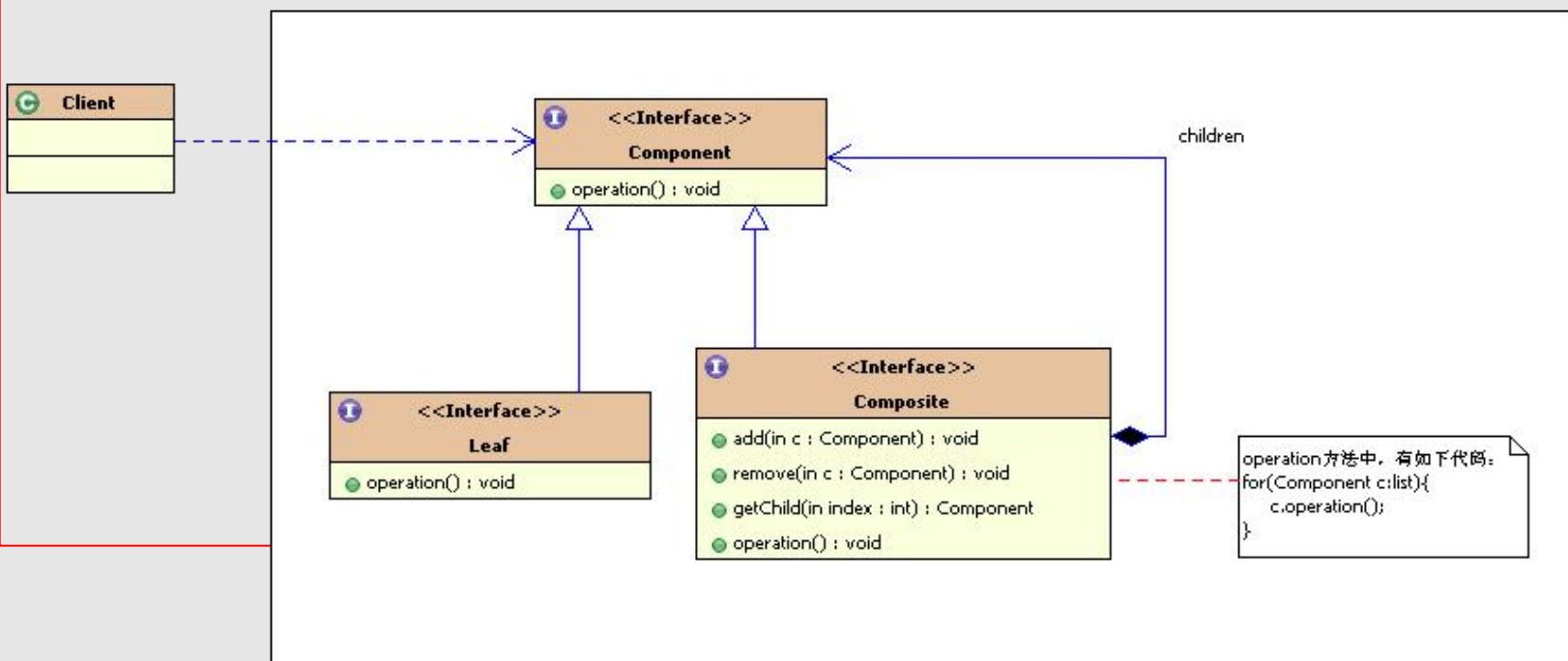
就像一个桥，将两个变化维度连接起来。
各个维度都可以独立的变化。
故称之为：桥模式

桥接模式

- 桥接模式实际开发中应用场景
 - JDBC驱动程序
 - AWT中的Peer架构
 - 银行日志管理：
 - 格式分类：操作日志、交易日志、异常日志
 - 距离分类：本地记录日志、异地记录日志
 - 人力资源系统中的奖金计算模块：
 - 奖金分类：个人奖金、团体奖金、激励奖金。
 - 部门分类：人事部门、销售部门、研发部门。
 - OA系统中的消息处理：
 - 业务类型：普通消息、加急消息、特急消息
 - 发送消息方式：系统内消息、手机短信、邮件

组合模式(composite)

- 使用组合模式的场景：
 - 把部分和整体的关系用树形结构来表示，从而使客户端可以使用统一的方式处理部分对象和整体对象。
- 组合模式核心：
 - 抽象构件(Component)角色：定义了叶子和容器构件的共同点
 - 叶子(Leaf)构件角色：无子节点
 - 容器(Composite)构件角色：有容器特征，可以包含子节点



组合模式(composite)

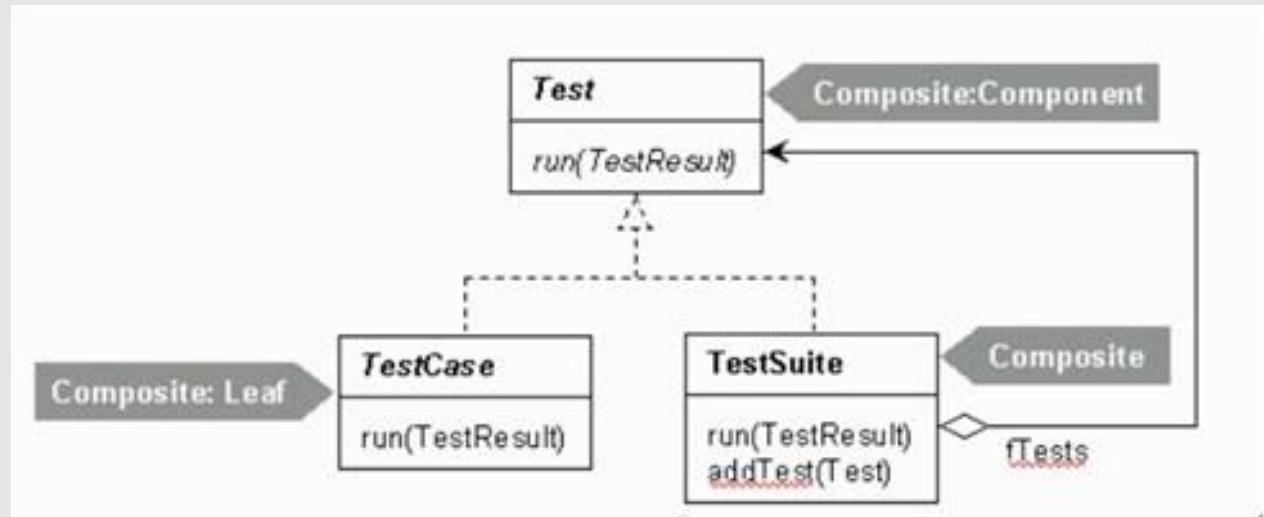
- 组合模式工作流程分析：
 - 组合模式为处理树形结构提供了完美的解决方案，描述了如何将容器和叶子进行递归组合，使得用户在使用时可以一致性的对待容器和叶子。
 - 当容器对象的指定方法被调用时，将遍历整个树形结构，寻找也包含这个方法的成员，并调用执行。其中，使用了递归调用的机制对整个结构进行处理。
- 使用组合模式，模拟杀毒软件架构设计

组合模式

- 开发中的应用场景：
 - 操作系统的资源管理器
 - GUI中的容器层次图
 - XML文件解析
 - OA系统中，组织结构的处理
 - Junit单元测试框架
 - 底层设计就是典型的组合模式，TestCase(叶子)、TestUnite(容器)、Test接口(抽象)

组合模式

- Junit单元测试框架底层设计
 - 底层设计就是典型的组合模式，TestCase(叶子)、TestSuite(容器)、Test接口(抽象)

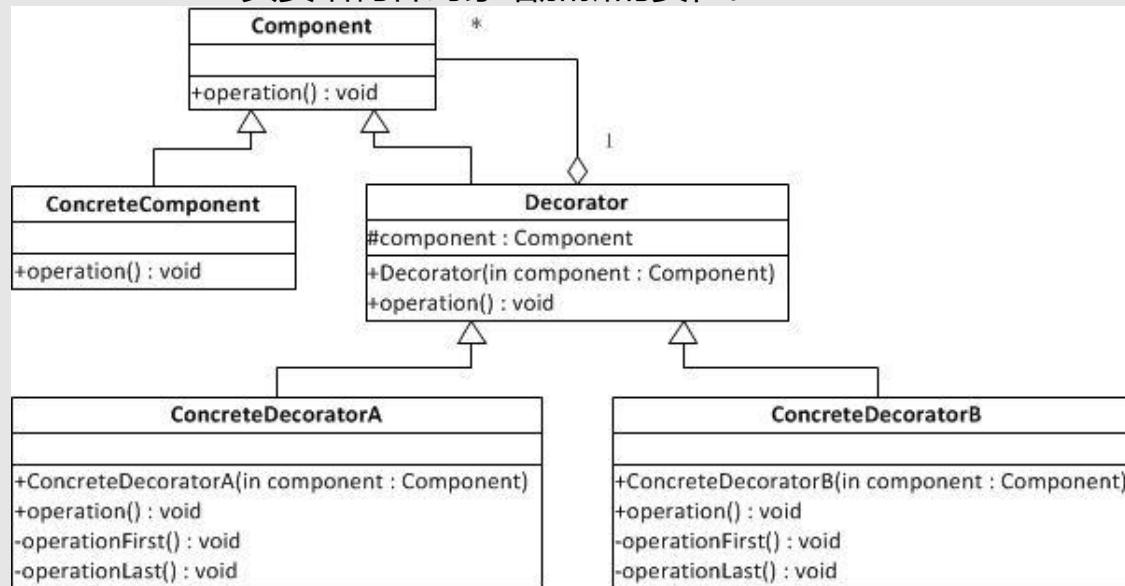


装饰模式(decorator)

- 职责：
 - 动态的为一个对象增加新的功能。
 - 装饰模式是一种用于代替继承的技术，**无须通过继承增加子类就能扩展对象的新功能**。使用对象的关联关系代替继承关系，更加灵活，**同时避免类型体系的快速膨胀**。

装饰模式(decorator)

- 实现细节：
 - Component抽象构件角色：
 - 真实对象和装饰对象有相同的接口。这样，客户端对象就能够以与真实对象相同的方式同装饰对象交互。
 - ConcreteComponent 具体构件角色(真实对象)：
 - io流中的FileInputStream、 FileOutputStream
 - Decorator装饰角色：
 - 持有一个抽象构件的引用。装饰对象接受所有客户端的请求，并把这些请求转发给真实的对象。这样，就能在真实对象调用前后增加新的功能。
 - ConcreteDecorator具体装饰角色：
 - 负责给构件对象增加新的责任。



装饰模式(decorator)

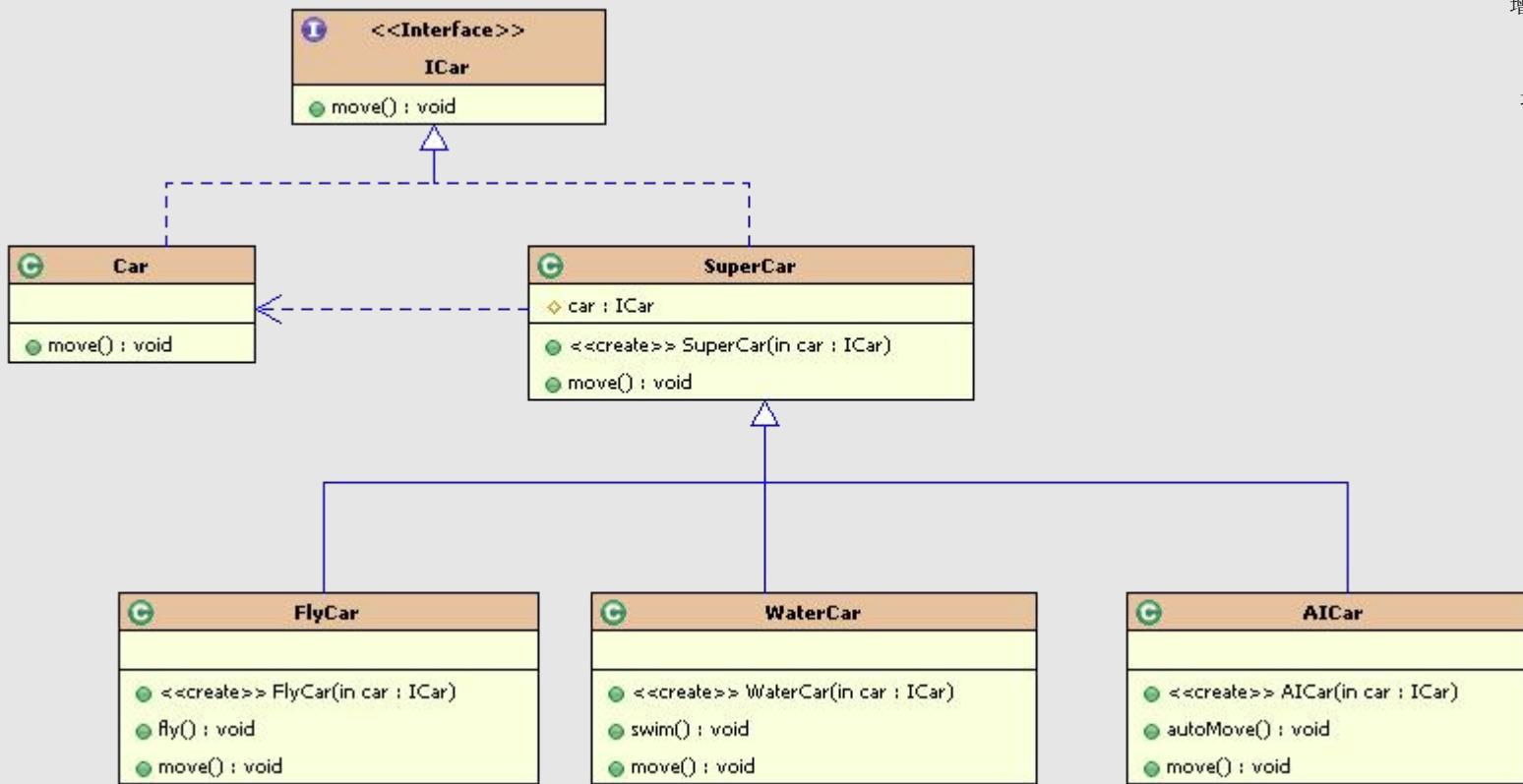
- 简单案例说明

增加人工智能，自动驾驶，汽车人

增加浮沉箱，水上汽车

增加翅膀，飞行汽车

车

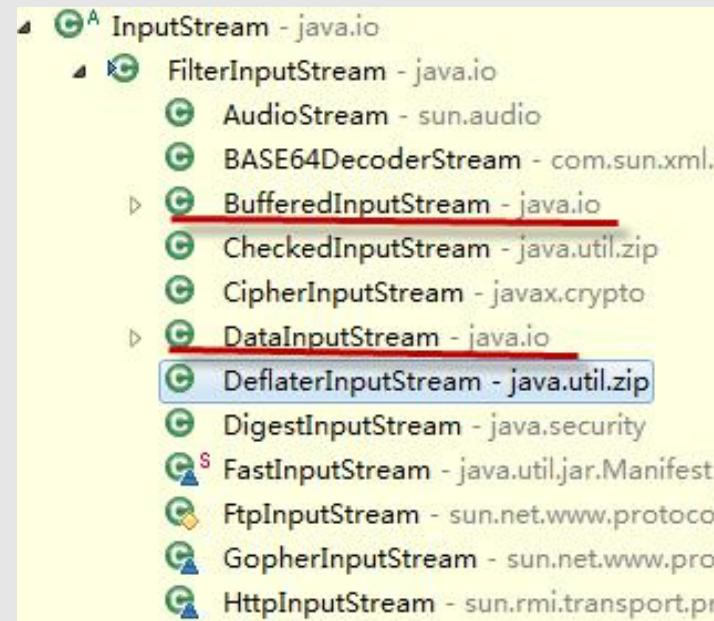
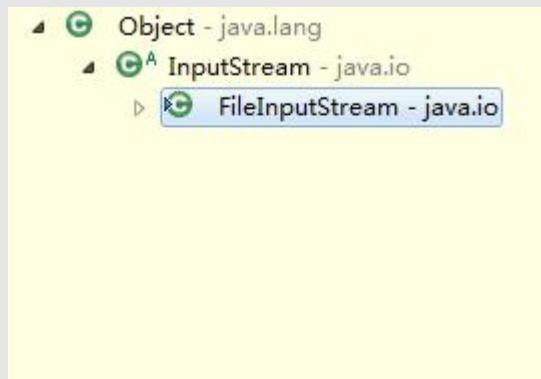


装饰模式(decorator)

- 开发中使用的场景：
 - IO中输入流和输出流的设计
 - Swing包中图形界面构件功能
 - Servlet API 中提供了一个request对象的Decorator设计模式的默认实现类HttpServletRequestWrapper , HttpServletRequestWrapper类，增强了request对象的功能。
 - Struts2中，request , response,session对象的处理

装饰模式(decorator)

- IO流实现细节：
 - Component抽象构件角色：
 - io流中的InputStream、OutputStream、Reader、Writer
 - ConcreteComponent 具体构件角色：
 - io流中的FileInputStream、 FileOutputStream
 - Decorator装饰角色：
 - 持有一个抽象构件的引用：io流中的FilterInputStream、 FilterOutputStream
 - ConcreteDecorator具体装饰角色：
 - 负责给构件对象增加新的责任。 Io流中的BufferedOutputStream、 BufferedInputStream等。



装饰模式(decorator)

- 总结：
 - 装饰模式（Decorator）也叫包装器模式（Wrapper）
 - 装饰模式降低系统的耦合度，可以动态的增加或删除对象的职责，并使得需要装饰的具体构建类和具体装饰类可以独立变化，以便增加新的具体构建类和具体装饰类。
- 优点
 - 扩展对象功能，比继承灵活，不会导致类个数急剧增加
 - 可以对一个对象进行多次装饰，创造出不同行为的组合，得到功能更加强大的对象
 - 具体构建类和具体装饰类可以独立变化，用户可以根据需要自己增加新的具体构件子类和具体装饰子类。
- 缺点
 - 产生很多小对象。大量小对象占据内存，一定程度上影响性能。
 - 装饰模式易于出错，调试排查比较麻烦。

装饰模式(decorator)

- 装饰模式和桥接模式的区别：
 - 两个模式都是为了解决过多子类对象问题。但他们的诱因不一样。桥模式是对象自身现有机制沿着多个维度变化，是既有部分不稳定。装饰模式是为了增加新的功能。

装饰模式(decorator)

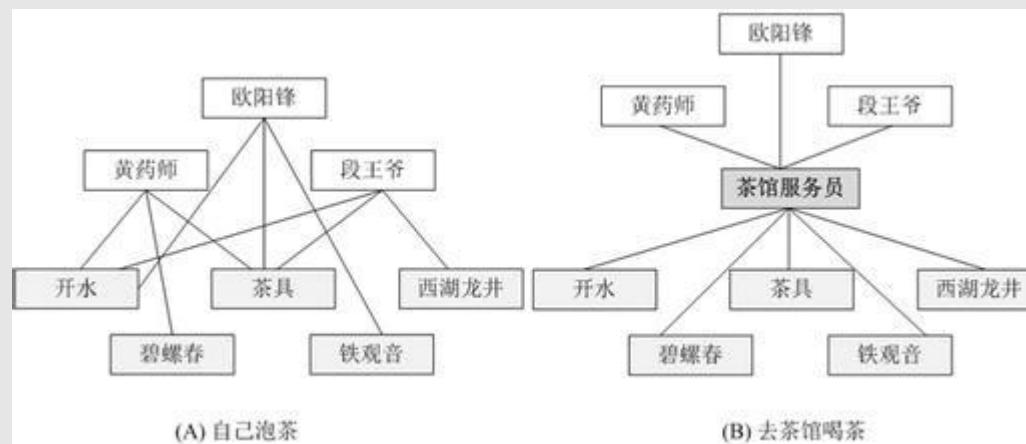
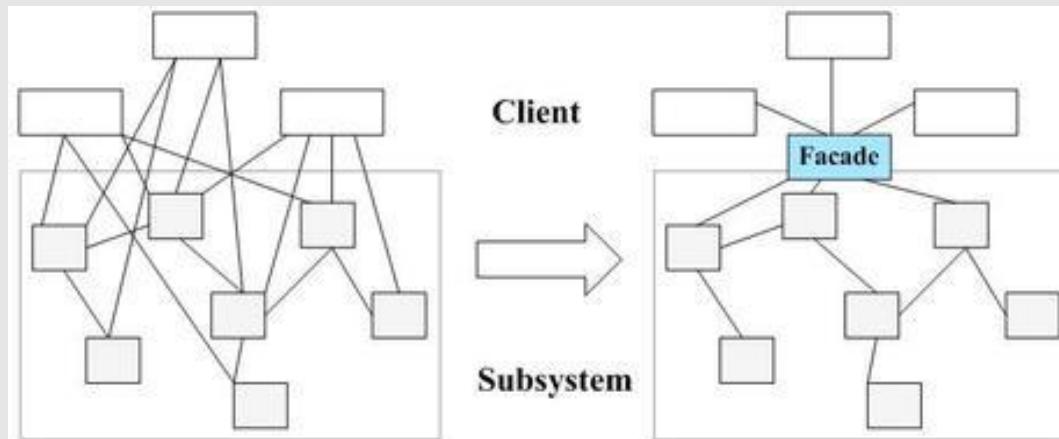
- 装饰模式和桥接模式的区别：
 - 两个模式都是为了解决过多子类对象问题。但他们的诱因不一样。桥模式是对象自身现有机制沿着多个维度变化，是既有部分不稳定。装饰模式是为了增加新的功能。

装饰模式(decorator)

- 装饰模式和桥接模式的区别：
 - 两个模式都是为了解决过多子类对象问题。但他们的诱因不一样。桥模式是对象自身现有机制沿着多个维度变化，是既有部分不稳定。装饰模式是为了增加新的功能。

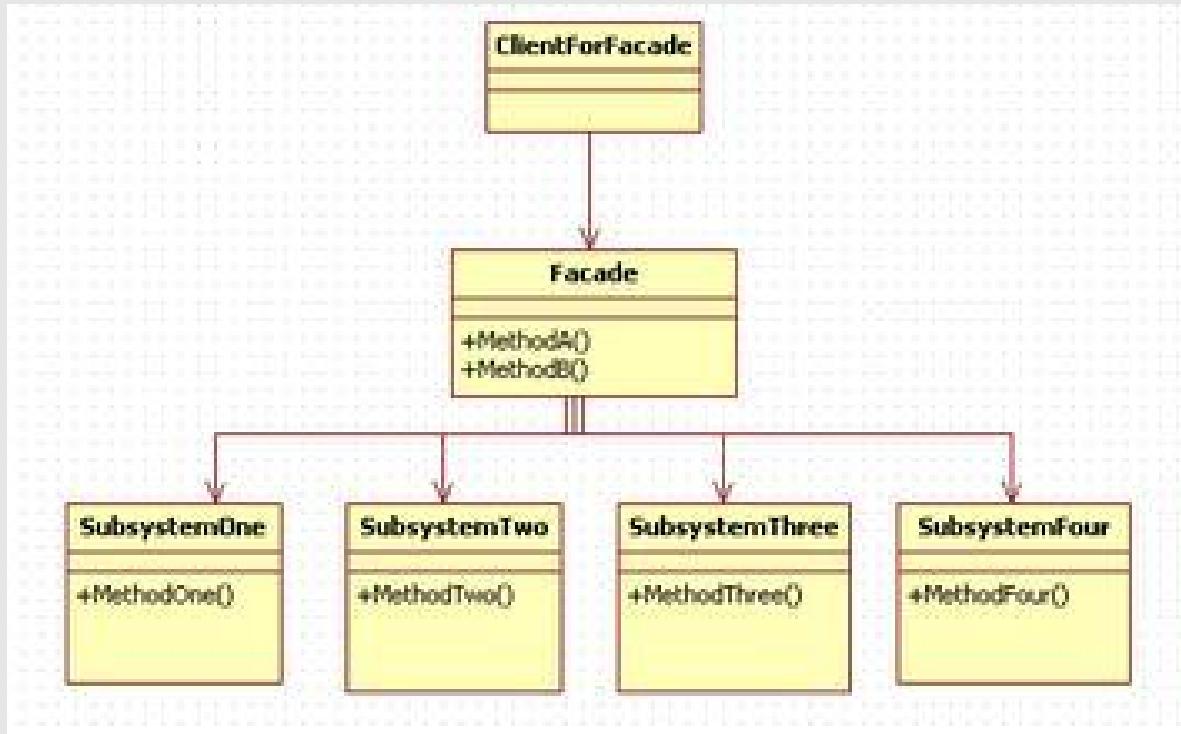
外观模式 facade

- 迪米特法则(最少知识原则)：
 - 一个软件实体应当尽可能少的与其他实体发生相互作用。



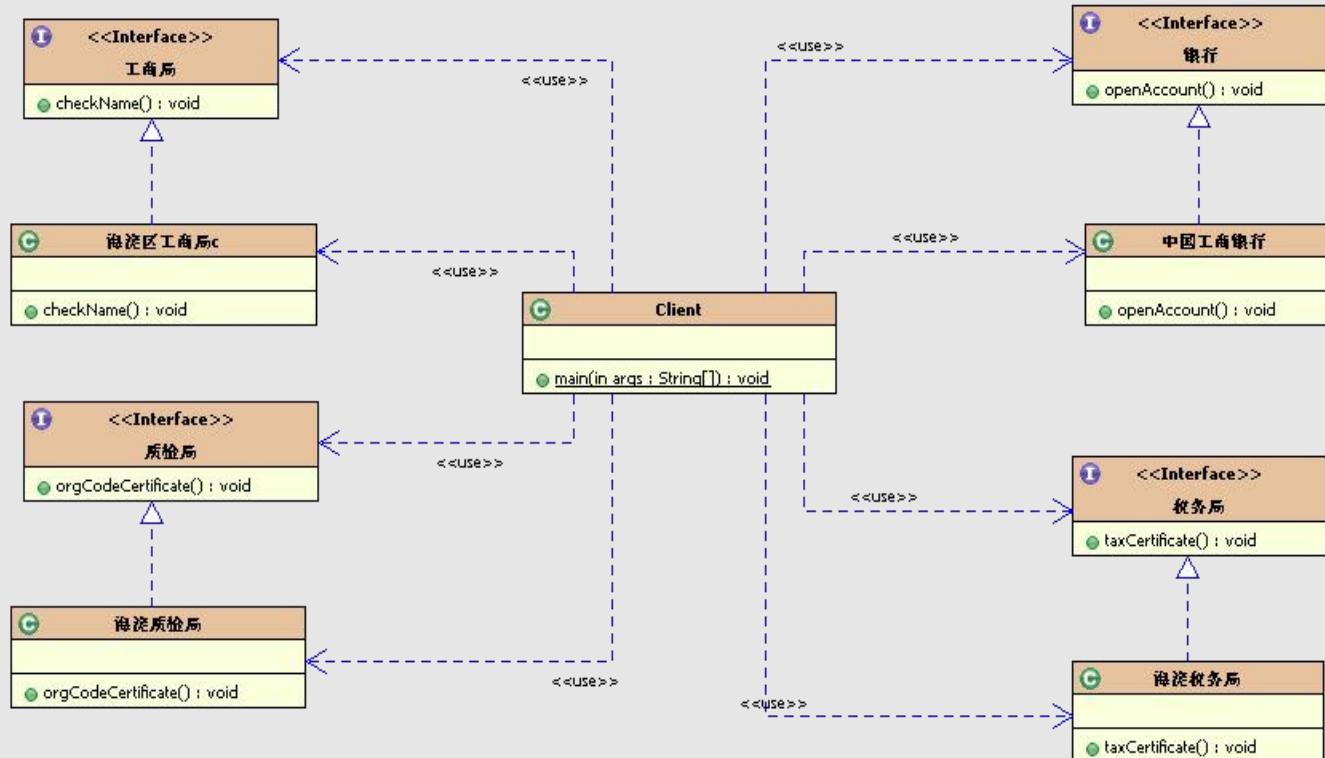
外观模式 facade

- 外观模式核心：
 - 为子系统提供统一的入口。封装子系统的复杂性，便于客户端调用。



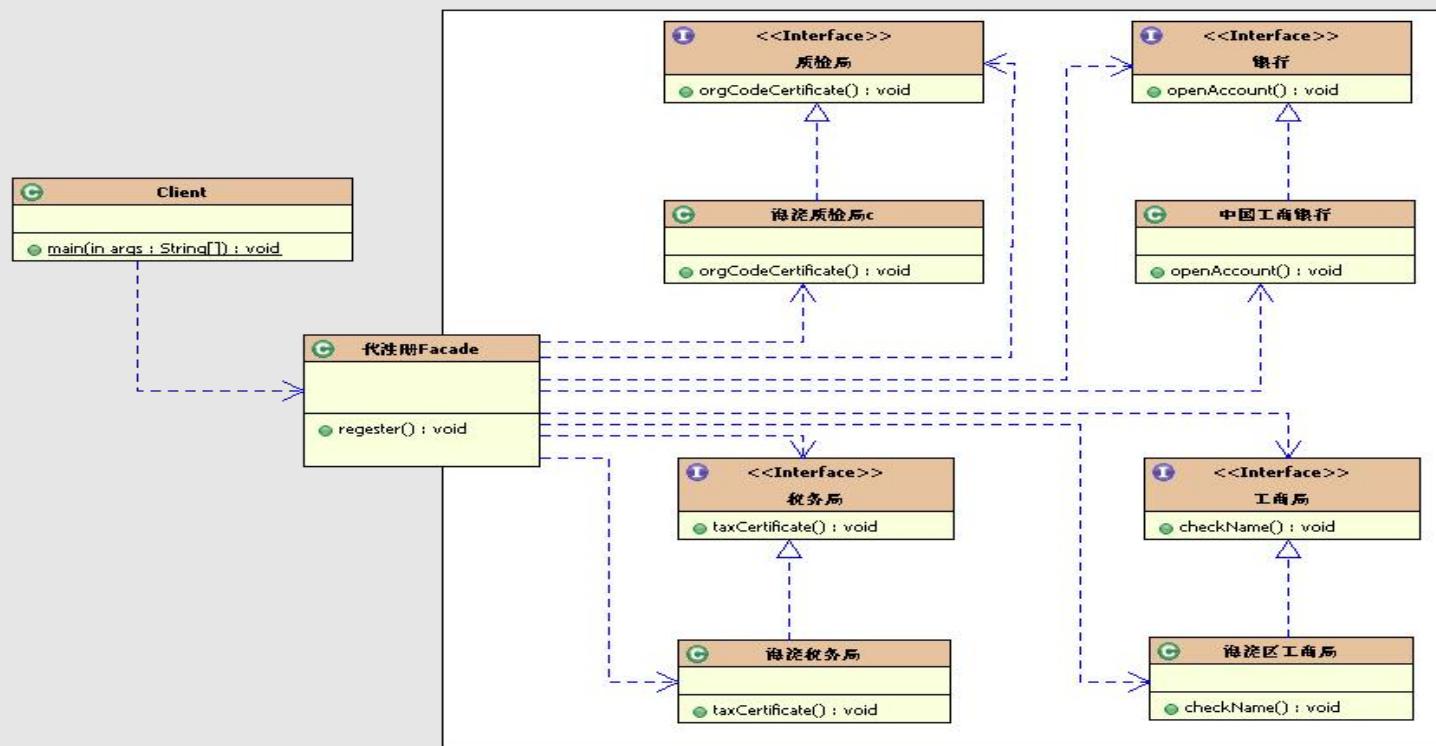
外观模式 facade

- 基本案例
 - 注册公司流程(不使用外观模式)



外观模式 facade

- 基本案例
 - 注册公司流程(使用外观模式)



外观模式 facade

- 开发中常见的场景
 - 频率很高。哪里都会遇到。各种技术和框架中，都有外观模式的使用。如：
 - JDBC封装后的，commons提供的DBUtils类，Hibernate提供的工具类、Spring JDBC工具类等

享元模式(FlyWeight)

- 场景：
 - 内存属于稀缺资源，不要随便浪费。如果有很多个完全相同或相似的对象，我们可以通过享元模式，节省内存。
- 核心：
 - 享元模式以共享的方式高效地支持大量细粒度对象的重用。
 - 享元对象能做到共享的关键是区分了内部状态和外部状态。
 - 内部状态：可以共享，不会随环境变化而改变
 - 外部状态：不可以共享，会随环境变化而改变

享元模式

- 围棋软件设计

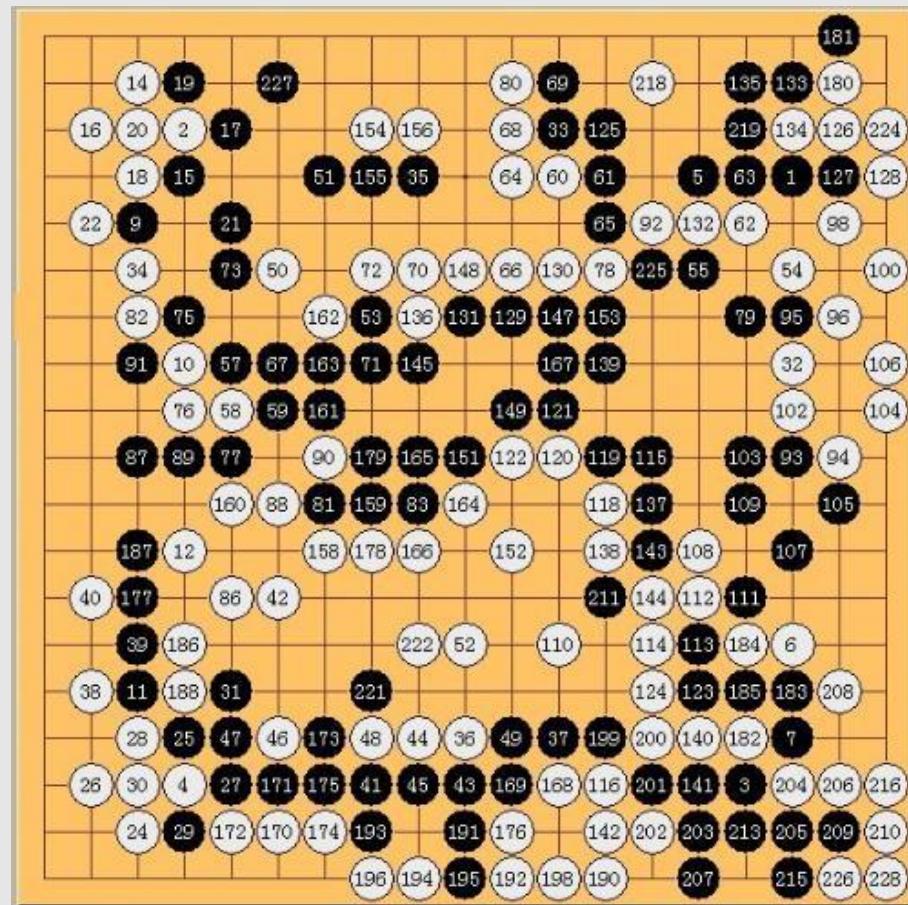
- 每个围棋棋子都是一个对象，
有如下属性：

颜色
形状
大小

(这些是可以共享的)
称之为：内部状态

位置

(这些不可以共享)
称之为：外部状态

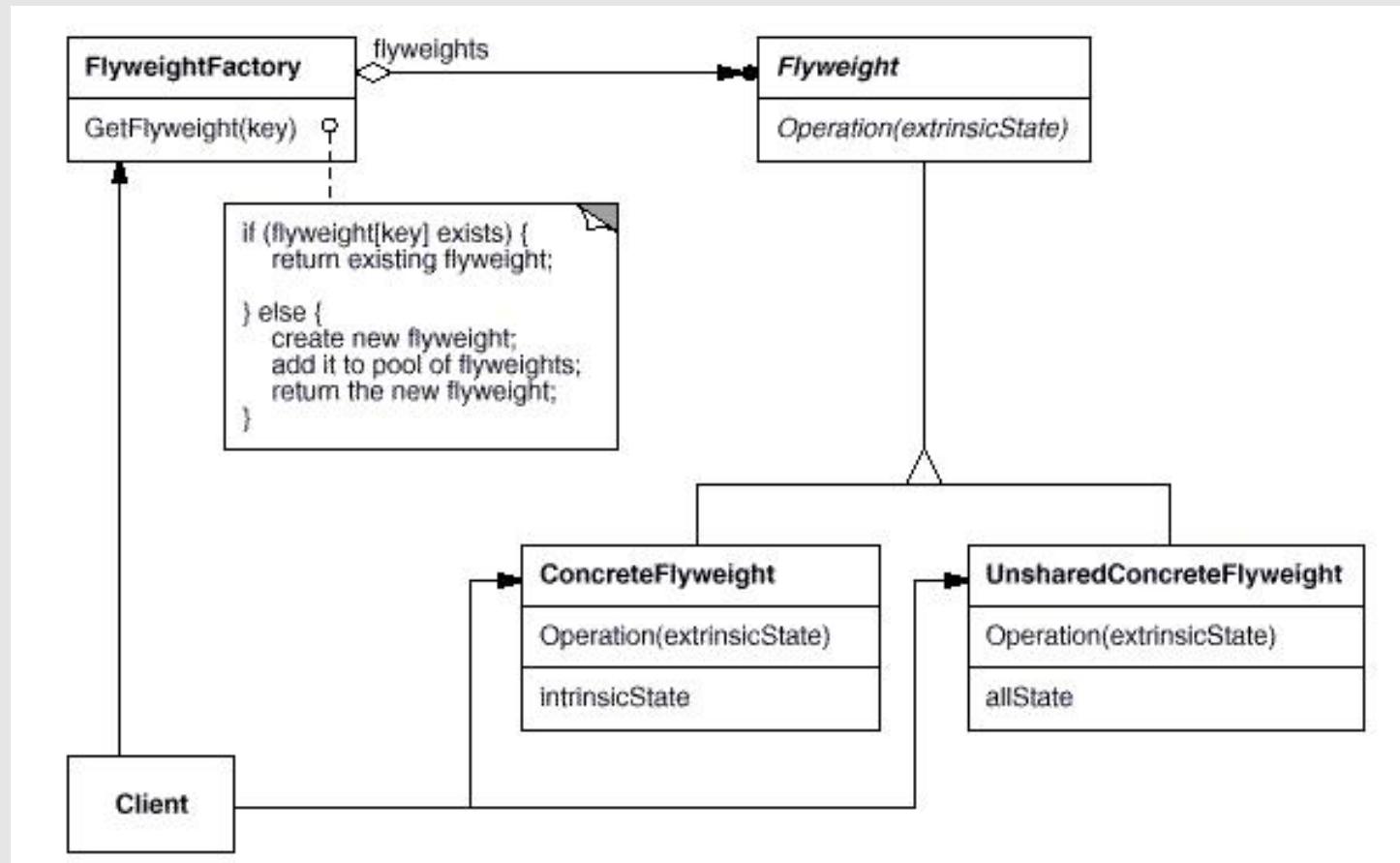


享元模式

- 享元模式实现：
 - FlyweightFactory享元工厂类
 - 创建并管理享元对象，享元池一般设计成键值对
 - FlyWeight抽象享元类
 - 通常是一个接口或抽象类，声明公共方法，这些方法可以向外界提供对象的内部状态，设置外部状态。
 - ConcreteFlyWeight具体享元类
 - 为内部状态提供成员变量进行存储
 - UnsharedConcreteFlyWeight非共享享元类
 - 不能被共享的子类可以设计为非共享享元类

享元模式

- 享元模式实现的UML图



享元模式

- 享元模式开发中应用的场景：
 - 享元模式由于其共享的特性，可以在任何“池”中操作，比如：线程池、数据库连接池。
 - String类的设计也是享元模式

享元模式

- 优点
 - 极大减少内存中对象的数量
 - 相同或相似对象内存中只存一份，极大的节约资源，提高系统性能
 - 外部状态相对独立，不影响内部状态
- 缺点
 - 模式较复杂，使程序逻辑复杂化
 - 为了节省内存，共享了内部状态，分离出外部状态，而读取外部状态使运行时间变长。用时间换取了空间。

结构型模式总结

• 结构型模式汇总

代理模式	为真实对象提供一个代理，从而控制对真实对象的访问
适配模式	使原本由于接口不兼容不能一起工作的类可以一起工作
桥接模式	处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各个维度可以独立的扩展在抽象层建立关联。
组合模式	将对象组合成树状结构以表示”部分和整体”层次结构，使得客户可以统一的调用叶子对象和容器对象
装饰模式	动态地给一个对象添加额外的功能，比继承灵活
外观模式	为子系统提供统一的调用接口，使得子系统更加容易使用
享元模式	运用共享技术有效的实现管理大量细粒度对象，节省内存，提高效率

行为型模式

- 行为型模式关注系统中对象之间的相互交互，研究系统在运行时对象之间的相互通信和协作，进一步明确对象的职责，共有11种模式。
- 创建型模式关注对象的创建过程。
- 结构型模式关注对象和类的组织。

行为型模式

• 行为型模式汇总：

- | | |
|----------|-------------------------|
| ① 责任链模式 | chain of responsibility |
| ② 命令模式 | command |
| ③ 解释器模式 | interpreter |
| ④ 迭代器模式 | iterator |
| ⑤ 中介者模式 | mediator |
| ⑥ 备忘录模式 | memento |
| ⑦ 观察者模式 | observer |
| ⑧ 状态模式 | state |
| ⑨ 策略模式 | strategy |
| ⑩ 模板方法模式 | template method |
| ⑪ 访问者模式 | visitor |

责任链模式chain of responsibility

- **定义：**

- 将能够处理同一类请求的对象连成一条链，所提交的请求沿着链传递，链上的对象逐个判断是否有能力处理该请求，如果能则处理，如果不能则传递给链上的下一个对象。

- **场景：**

- 打牌时，轮流出牌
- 接力赛跑
- 大学中，奖学金审批
- 公司中，公文审批



责任链模式chain of responsibility

- 场景：
 - 公司里面，报销个单据需要经过流程：
 - 申请人填单申请，申请给经理
 - 小于1000，经理审查。
 - 超过1000，交给总经理审批。
 - 总经理审批通过
 - 公司里面，请假条的审批过程：
 - 如果请假天数小于3天，主任审批
 - 如果请假天数大于等于3天，小于10天，经理审批
 - 如果大于等于10天，小于30天，总经理审批
 - 如果大于等于30天，提示拒绝

责任链模式chain of responsibility

- 场景：
 - 公司里面，报销个单据需要经过流程：
 - 申请人填单申请，申请给经理
 - 小于1000，经理审查。
 - 超过1000，交给总经理审批。
 - 总经理审批通过
 - 公司里面，请假条的审批过程：
 - 如果请假天数小于3天，主任审批
 - 如果请假天数大于等于3天，小于10天，经理审批
 - 如果大于等于10天，小于30天，总经理审批
 - 如果大于等于30天，提示拒绝

责任链模式chain of responsibility

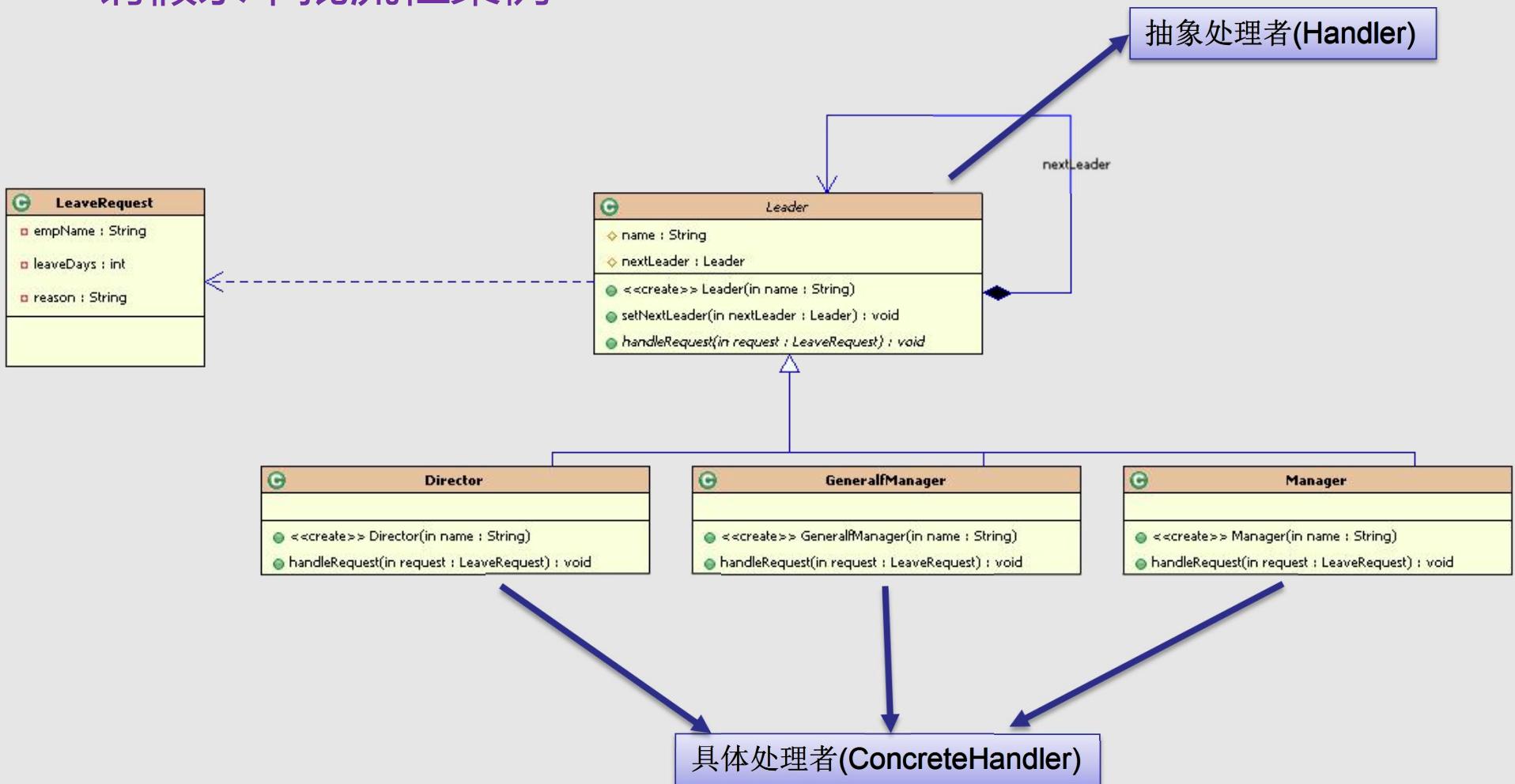
- 场景：
 - 公司里面，报销个单据需要经过流程：
 - 申请人填单申请，申请给经理
 - 小于1000，经理审查。
 - 超过1000，交给总经理审批。
 - 总经理审批通过
 - 公司里面，请假条的审批过程：
 - 如果请假天数小于3天，主任审批
 - 如果请假天数大于等于3天，小于10天，经理审批
 - 如果大于等于10天，小于30天，总经理审批
 - 如果大于等于30天，提示拒绝

责任链模式chain of responsibility

- 场景：
 - 公司里面，SCM(Supply Chain Management供应链管理)系统中，采购审批子系统的设计：
 - 采购金额小于5万，主任审批
 - 采购金额大于等于5万，小于10万，经理审批
 - 采购金额大于等于10万，小于20万，副总经理审批
 - 采购金额大于等于20万，总经理审批

责任链模式chain of responsibility

- 请假条审批流程案例



责任链模式chain of responsibility

- 添加新的处理对象：
 - 由于责任链的创建完全在客户端，因此新增新的具体处理者对原有类库没有任何影响，只需添加新的类，然后在客户端调用时添加即可。符合开闭原则。
 - 案例：
 - 我们可以在请假处理流程中，增加新的“副总经理”角色，审批大于等于10天，小于20天的请假。审批流程变为：
 - ① 如果请假天数小于3天，主任审批
 - ② 如果请假天数大于等于3天，小于10天，经理审批
 - ③ **大于等于10天，小于20天的请假，副总经理审批**
 - ④ 如果大于等于20天，小于30天，总经理审批
 - ⑤ 如果大于等于30天，提示拒绝

责任链模式chain of responsibility

- 链表方式定义职责链(上一个案例)
- 非链表方式实现职责链
 - 通过集合、数组生成职责链更加实用！实际上，很多项目中，每个具体的Handler并不是由开发团队定义的，而是项目上线后由外部单位追加的，所以使用链表方式定义COR链就很困难。

责任链模式chain of responsibility

- 开发中常见的场景：
 - Java中，异常机制就是一种责任链模式。一个try可以对应多个catch，当第一个catch不匹配类型，则自动跳到第二个catch.
 - Javascript语言中，事件的冒泡和捕获机制。Java语言中，事件的处理采用观察者模式。
 - Servlet开发中，过滤器的链式处理
 - Struts2中，拦截器的调用也是典型的责任链模式

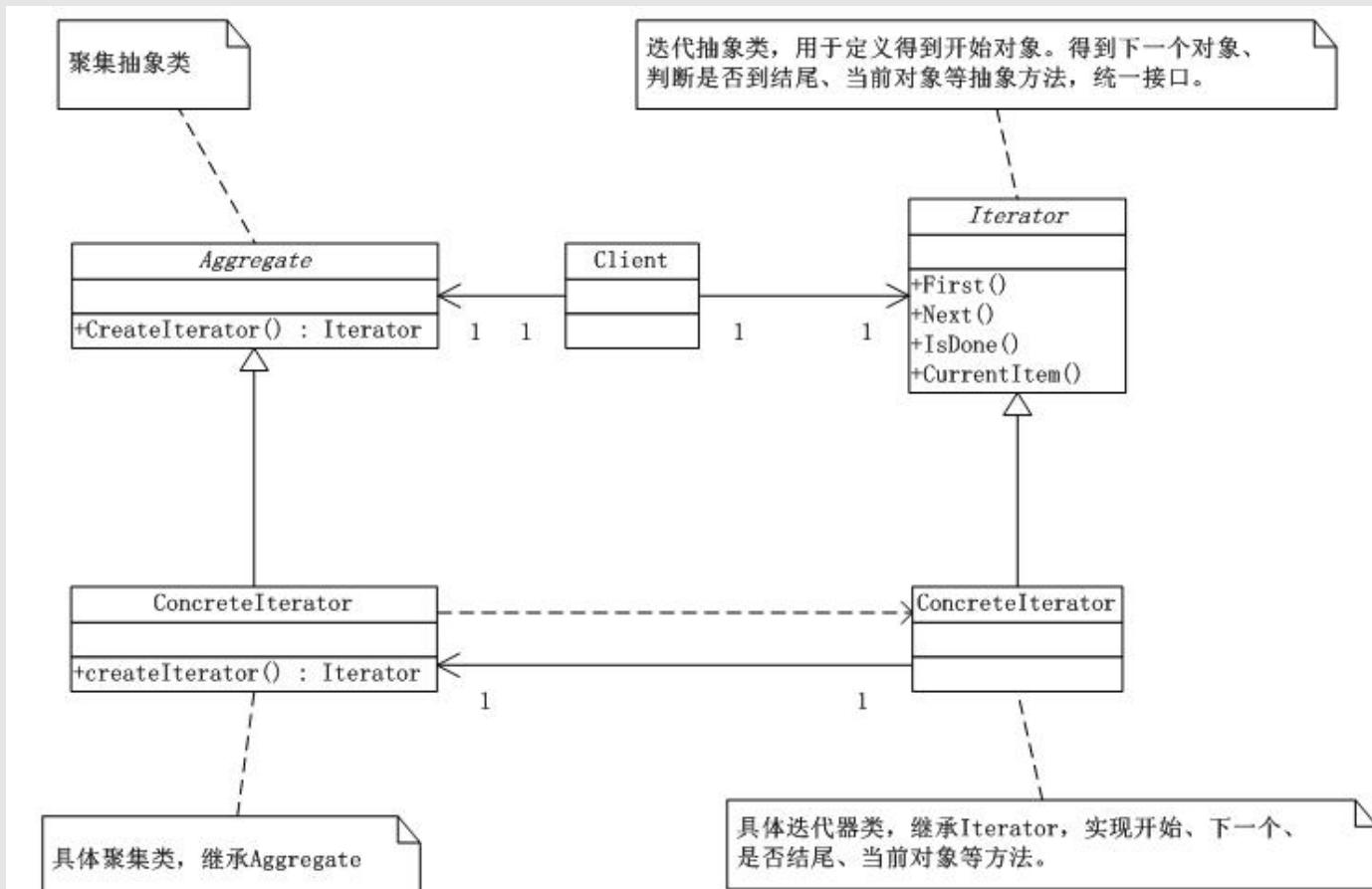
迭代器模式 iterator

- 场景：
 - 提供一种可以遍历聚合对象的方式。又称为：游标cursor模式
 - 聚合对象：存储数据
 - 迭代器：遍历数据



迭代器模式iterator

- 结构：
 - 聚合对象：存储数据
 - 迭代器：遍历数据



迭代器模式iterator

- 基本案例：
 - 实现正向遍历的迭代器
 - 实现逆向遍历的迭代器
- 开发中常见的场景：
 - JDK内置的迭代器(List/Set)

迭代器模式iterator

- 基本案例：
 - 实现正向遍历的迭代器
 - 实现逆向遍历的迭代器
- 开发中常见的场景：
 - JDK内置的迭代器(List/Set)

中介者模式 Mediator

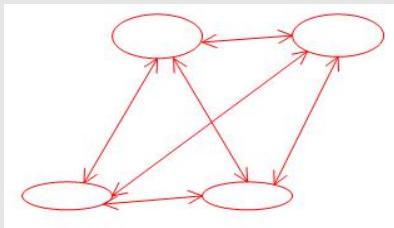
- 中介大家熟悉吗？



中介者模式 Mediator

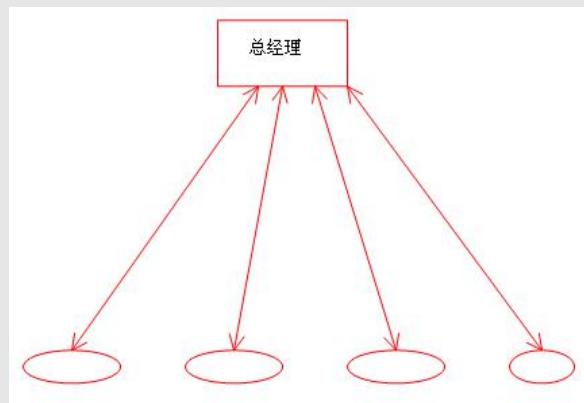
- 场景(中介大家熟悉吗?房产中介?)：

- 假如没有总经理。下面三个部门：财务部、市场部、研发部。财务部要发工资，让大家核对公司需要跟市场部和研发部都通气；市场部要接个新项目，需要研发部处理技术、需要财务部出资金。市场部跟各个部门打交道。 虽然只有三个部门，但是关系非常乱。



- 实际上，公司都有总经理。各个部门有什么事情都通报到总经理这里，总经理再通知各个相关部门。

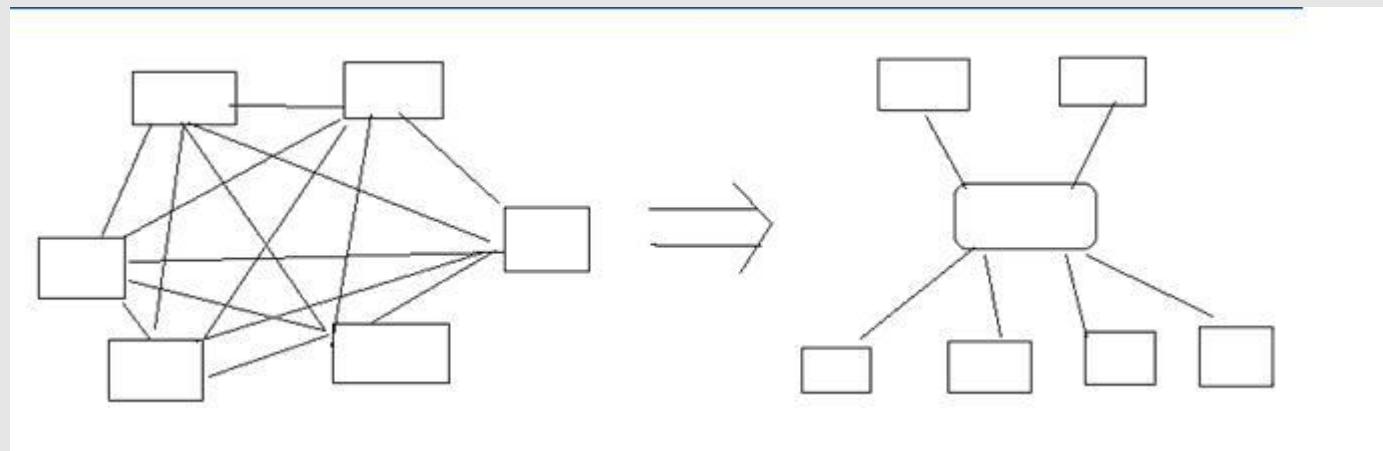
- 这就是一个典型的“中介者模式”
总经理起到一个中介、协调的作用



中介者模式 Mediator

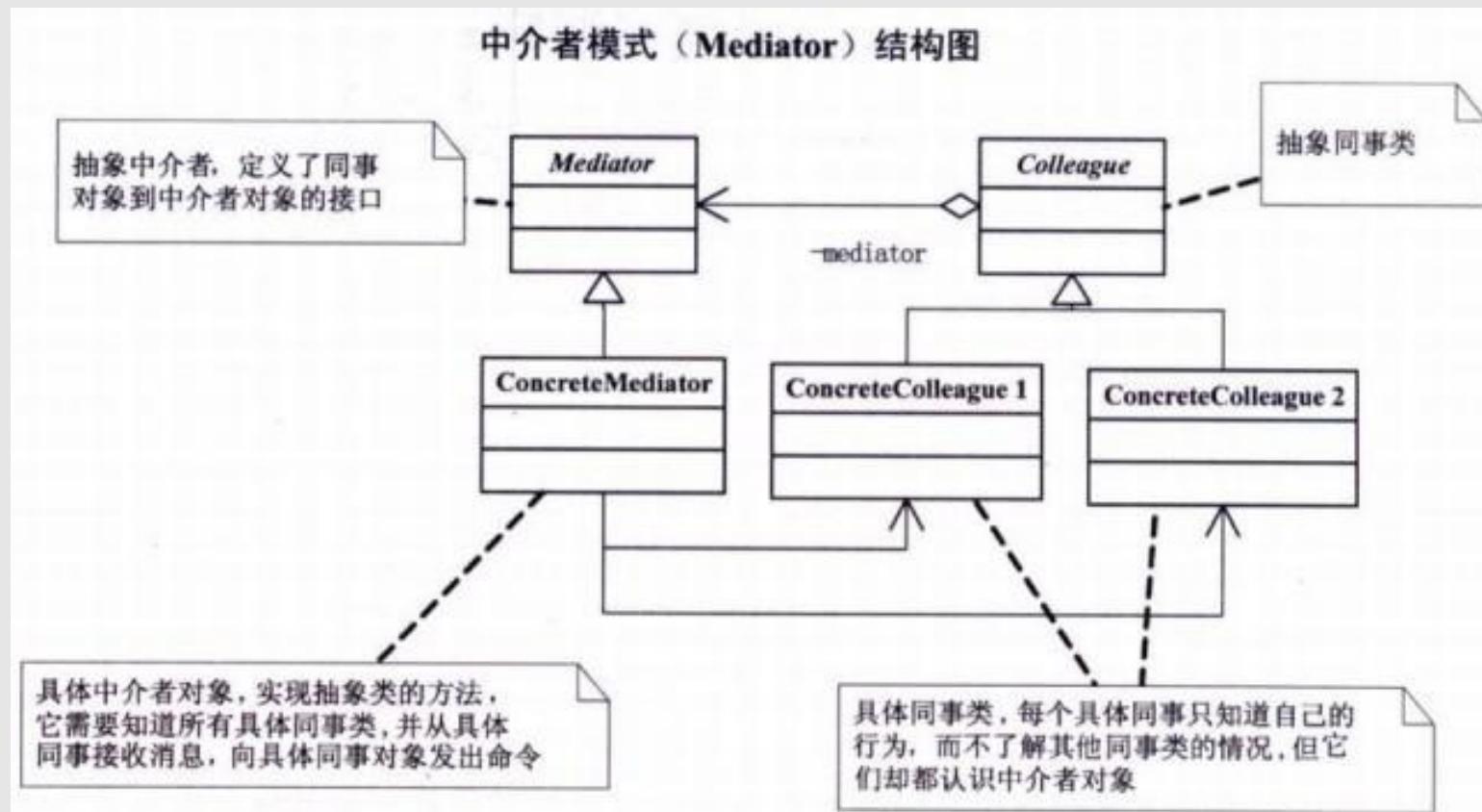
- 核心：

- 如果一个系统中对象之间的联系呈现为网状结构，对象之间存在大量多对多关系，将导致关系及其复杂，**这些对象称为“同事对象”**
- 我们可以引入**一个中介者对象**，使各个同事对象只跟中介者对象打交道，将复杂的网络结构化解为如下的星形结构。



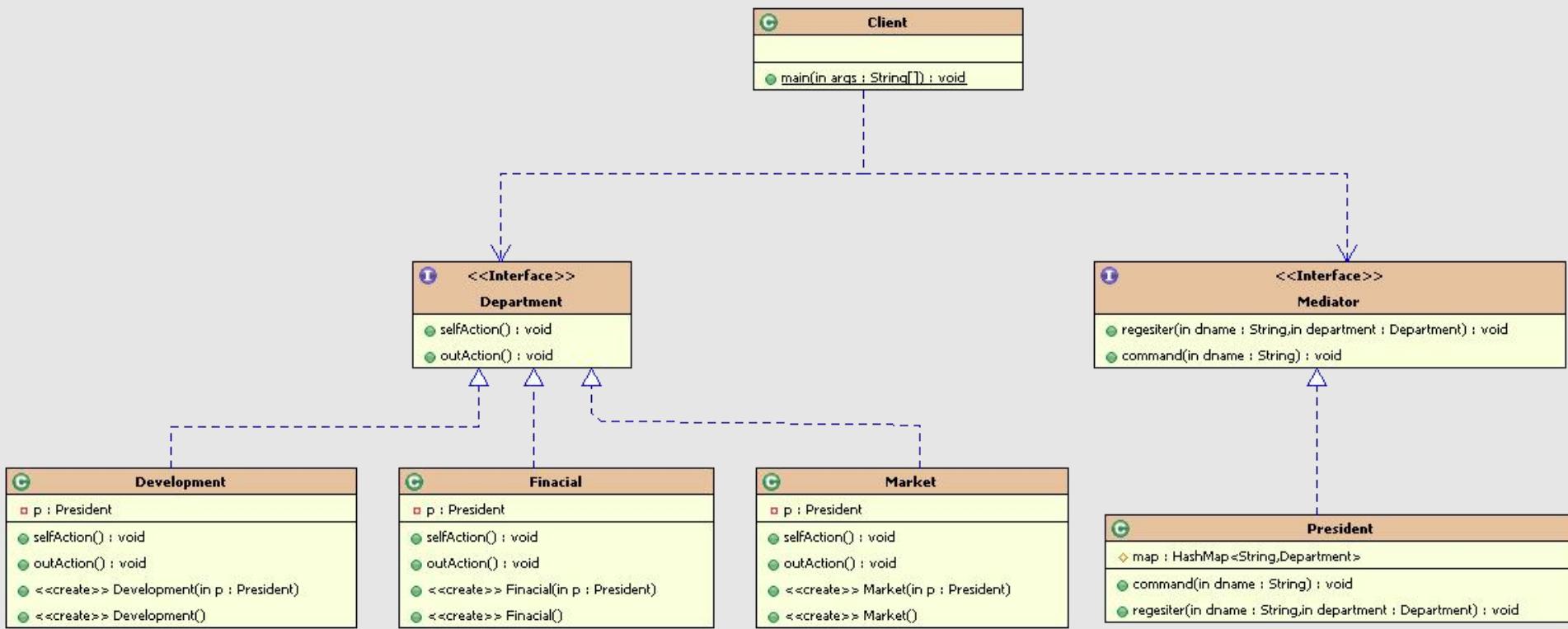
中介者模式 Mediator

• 中介者模式类图



中介者模式 Mediator

- 课堂代码类图



中介者模式 Mediator

- **中介者模式的本质：**
 - 解耦多个同事对象之间的交互关系。每个对象都持有中介者对象的引用，只跟中介者对象打交道。我们通过中介者对象统一管理这些交互关系
- **开发中常见的场景：**
 - MVC模式(其中的C，控制器就是一个中介者对象。M和V都和他打交道)
 - 窗口游戏程序，窗口软件开发中窗口对象也是一个中介者对象
 - 图形界面开发GUI中，多个组件之间的交互，可以通过引入一个中介者对象来解决，可以是整体的窗口对象或者DOM对象
 - Java.lang.reflect.Method#invoke()

命令模式 command

• 介绍：

- 命令模式：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。也称之为：动作Action模式、事务transaction模式

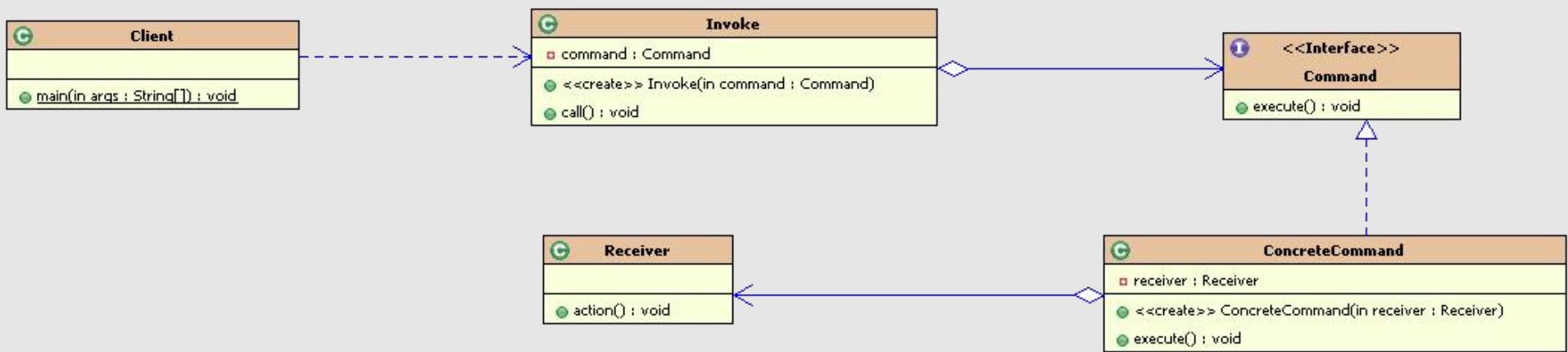


命令模式 command

- 结构：
 - Command抽象命令类
 - ConcreteCommand具体命令类
 - Invoker调用者/请求者
 - 请求的发送者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联。在程序运行时，将调用命令对象的execute()，间接调用接收者的相关操作。
 - Receiver接收者
 - 接收者执行与请求相关的操作，具体实现对请求的业务处理。
 - 未抽象前，实际执行操作内容的对象。
 - Client客户类
 - 在客户类中需要创建调用者对象、具体命令类对象，在创建具体命令对象时指定对应的接收者。发送者和接收者之间没有直接关系，都通过命令对象间接调用。

命令模式 command

- 相关类的类图



命令模式 command

- 开发中常见的场景：
 - Struts2中，action的整个调用过程中就有命令模式。
 - 数据库事务机制的底层实现
 - 命令的撤销和恢复

解释器模式 Interpreter

- 介绍：
 - 是一种不常用的设计模式
 - 用于描述如何构成一个简单的语言解释器，主要用于使用面向对象语言开发的编译器和解释器设计。
 - 当我们需要开发一种新的语言时，可以考虑使用解释器模式。
 - 尽量不要使用解释器模式，后期维护会有很大麻烦。在项目中，可以使用Jruby，Groovy、java的js引擎来替代解释器的作用，弥补java语言的不足。



解释器模式 Interpreter

- 开发中常见的场景：
 - EL表达式式的处理
 - 正则表达式解释器
 - SQL语法的解释器
 - 数学表达式解析器
 - 如现成的工具包:Math Expression String Parser、Expression4J等。
 - MESP的网址：<http://sourceforge.net/projects/expression-tree/>
 - Expression4J的网址：<http://sourceforge.net/projects/expression4j/>

访问者模式 Visitor

- 模式动机：
 - 对于存储在一个集合中的对象，他们可能具有不同的类型(即使有一个公共的接口)，对于该集合中的对象，可以接受一类称为访问者的对象来访问，不同的访问者其访问方式也有所不同。
- 定义：
 - 表示一个作用于某对象结构中的各元素的操作，它使我们可以在不改变个元素的类的前提下定义作用于这些元素的新操作。
- **开发中的场景(应用范围非常窄，了解即可)：**
 - XML文档解析器设计
 - 编译器的设计
 - 复杂集合对象的处理

策略模式 strategy

- 场景：
 - 某个市场人员接到单后的报价策略(CRM系统中常见问题)。报价策略很复杂，可以简单作如下分类：
 - 普通客户小批量报价
 - 普通客户大批量报价
 - 老客户小批量报价
 - 老客户大批量报价
 - 具体选用哪个报价策略，这需要根据实际情况来确定。这时候，我们采用策略模式即可。



策略模式 strategy

- 我们先可以采用条件语句处理：

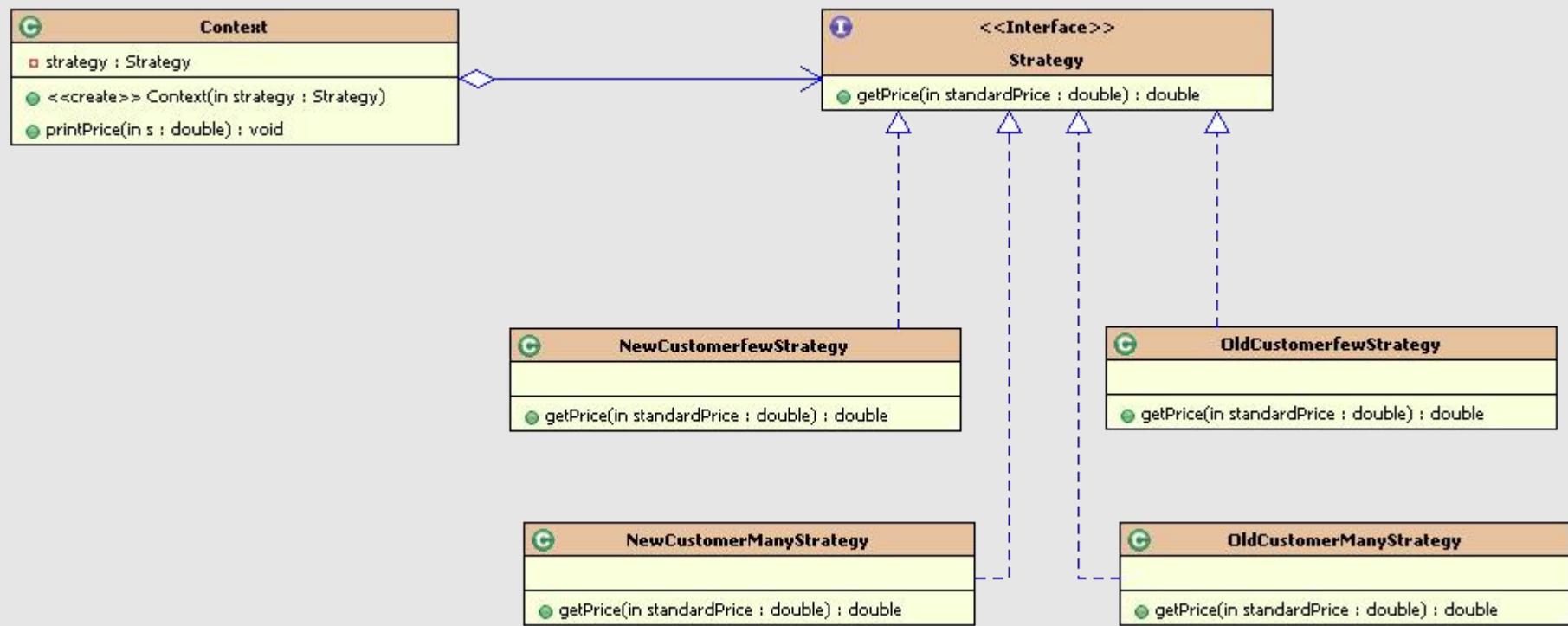
```
public double getPrice(String type, double price){  
  
    if(type.equals("普通客户小批量")){  
        System.out.println("不打折,原价");  
        return price;  
    }else if(type.equals("普通客户大批量")){  
        System.out.println("打九折");  
        return price*0.9;  
    }else if(type.equals("老客户小批量")){  
        System.out.println("打八五折");  
        return price*0.85;  
    }else if(type.equals("老客户大批量")){  
        System.out.println("打八折");  
        return price*0.8;  
    }  
    return price;  
}
```

实现起来比较容易，
符合一般开发人员的思路

- 假如，类型特别多，算法比较复杂时，整个条件控制代码会变得很长，难于维护。

策略模式 strategy

- 策略模式
 - 策略模式对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便的更换算法或者增加新的算法。并且由客户端决定调用哪个算法。
- 课堂案例的类图

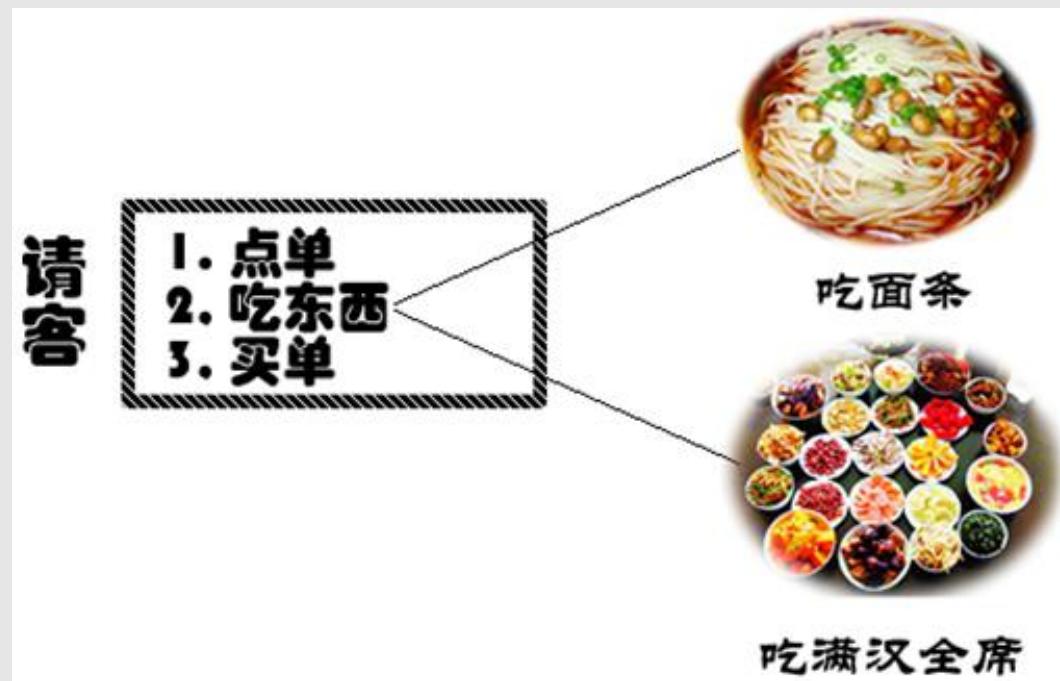


策略模式 strategy

- 本质：
 - 分离算法，选择实现。
- 开发中常见的场景：
 - JAVASE中GUI编程中，布局管理
 - Spring框架中，Resource接口，资源访问策略
 - javax.servlet.http.HttpServlet#service()

模板方法模式 template method

- 场景：
 - 客户到银行办理业务：
 - ① 取号排队
 - ② 办理具体现金/转账/企业/个人/理财业务
 - ③ 给银行工作人员评分



模板方法模式 template method

- 模板方法模式介绍：
 - 模板方法模式是编程中经常用得到模式。它定义了一个操作中的算法骨架，将某些步骤延迟到子类中实现。这样，新的子类可以在不改变一个算法结构的前提下重新定义该算法的某些特定步骤。
- 核心：
 - 处理某个流程的代码已经都具备，但是其中某个节点的代码暂时不能确定。因此，我们采用工厂方法模式，将这个节点的代码实现转移给子类完成。**即：处理步骤父类中定义好，具体实现延迟到子类中定义**

模板方法模式 template method

- 案例代码

```
public abstract class BankTemplateMethod {  
    //具体方法  
    public void takeNumber(){  
        System.out.println("取号排队");  
    }  
  
    public abstract void transact(); //办理具体的业务 //钩子方法  
  
    public void evaluate(){  
        System.out.println("反馈评分");  
    }  
  
    //模板方法, 把基本操作组合到一起, 子类一般不能重写  
    public final void process(){  
        this.takeNumber();  
  
        this.transact();  
  
        this.evaluate();  
    }  
}
```

模板方法, 把基本操作组合到一起, 子类一般不能重写

像个钩子。执行时,
挂哪个子类的方法
就调用哪个

模板方法模式 template method

- 方法回调(钩子方法)

- 好莱坞原则” Don' t call me , we' ll call you back ”

- 在好莱坞，当**艺人**把简历递交给好莱坞的**娱乐公司**时，所能做的就是等待，整个过程由娱乐公司控制，演员只能被动地服务安排，在需要的时候再由公司安排具体环节的演出。

- 在软件开发中，我们可以将call翻译为调用。**子类**不能调用**父类**，而通过父类调用子类。这些调用步骤已经在父类中写好了，完全由父类控制整个过程。

模板方法模式 template method

- **什么时候用到模板方法模式：**

- 实现一个算法时，整体步骤很固定。但是，某些部分易变。易变部分可以抽象成出来，供子类实现。

- **开发中常见的场景：**

- 非常频繁。各个框架、类库中都有他的影子。比如常见的有：
 - 数据库访问的封装
 - Junit单元测试
 - servlet中关于doGet/doPost方法调用
 - Hibernate中模板程序
 - spring中JDBCTemplate、HibernateTemplate等。

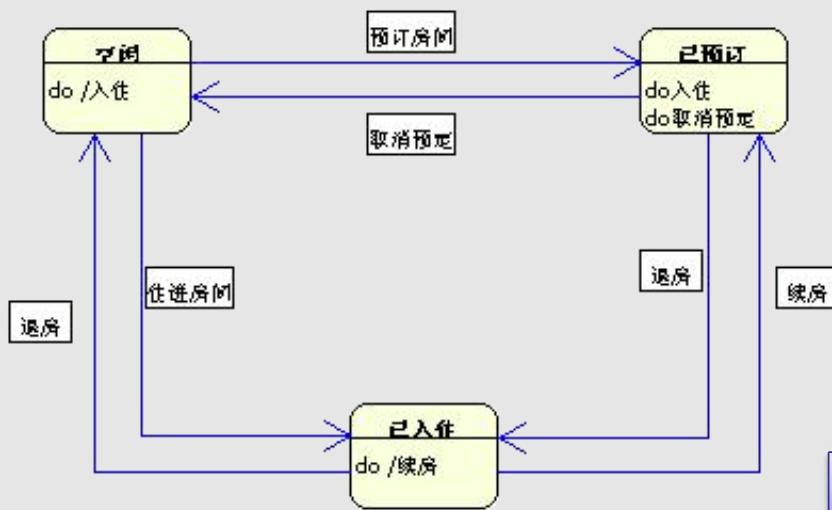
状态模式 state

- 场景：
 - 电梯的运行
 - 维修、正常、自动关门、自动开门、向上运行、向下运行、消防状态
 - 红绿灯
 - 红灯、黄灯、绿灯
 - 企业或政府系统
 - 公文的审批状态
 - 报销单据审批状态
 - 假条审批
 - 网上购物时，订单的状态
 - 下单
 - 已付款
 - 已发货
 - 送货中
 - 已收货



状态模式 state

- 场景：
 - 酒店系统中，房间的状态变化：
 - 已预订
 - 已入住
 - 空闲

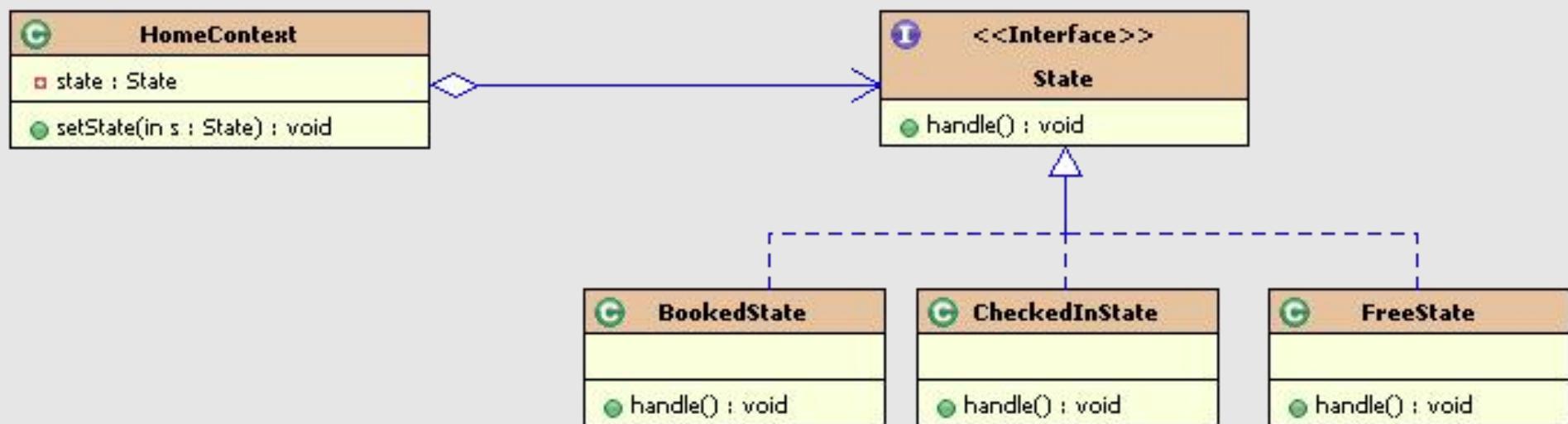


```
if(state=="空闲"){
    if(预订房间){
        预定操作;
        state="已预订";
    }else if(住进房间){
        入住操作;
        state="已入住";
    }
}else if("已预订"){
    if(住进房间){
        入住操作;
        state="已入住";
    }else if(取消预订){
        取消操作;
        state="空闲";
    }
}
```

当遇到这种需要频繁的修改状态时，考虑状态模式

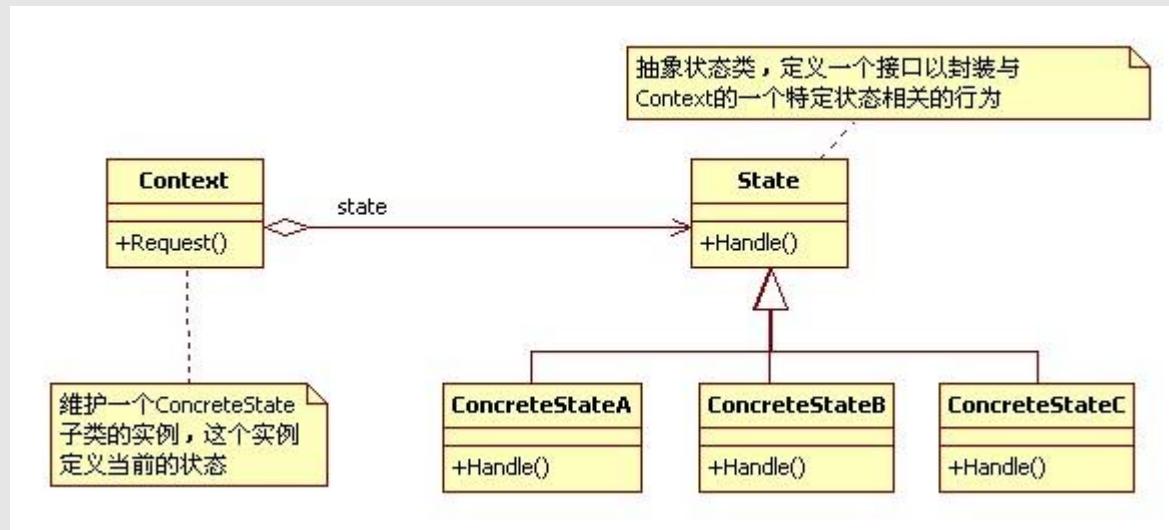
状态模式 state

- 酒店系统中，房间状态变化的相关类图结构：



状态模式 state

- 核心：
 - 用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题
- 结构：
 - Context环境类
 - 环境类中维护一个State对象，他是定义了当前的状态。
 - State抽象状态类
 - ConcreteState具体状态类
 - 每一个类封装了一个状态对应的行为

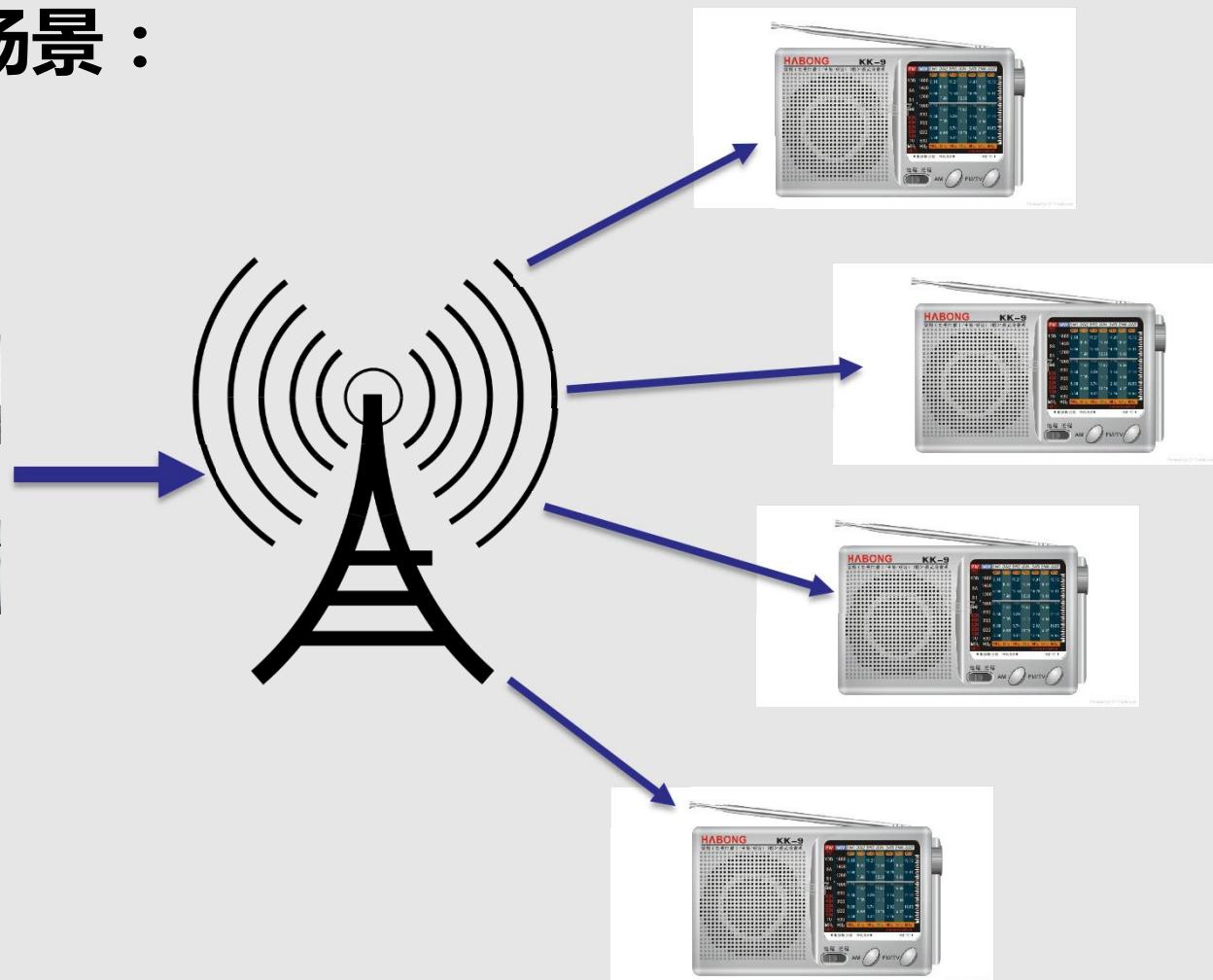


状态模式 state

- 开发中常见的场景：
 - 银行系统中账号状态的管理
 - OA系统中公文状态的管理
 - 酒店系统中，房间状态的管理
 - 线程对象各状态之间的切换

观察者模式 Observer

- 广播机制的场景：



观察者模式 Observer

• 场景：

- 聊天室程序的创建。服务器创建好后，A,B,C三个客户端连上来公开聊天。A向服务器发送数据，服务器端聊天数据改变。我们希望将这些聊天数据分别发给其他在线的客户。也就是说，每个客户端需要更新服务器端得数据。
- 网站上，很多人订阅了“java主题”的新闻。当有这个主题新闻时，就会将这些新闻发给所有订阅的人。
- 大家一起玩CS游戏时，服务器需要将每个人方位变化发给所有的客户。

上面这些场景，我们都可以使用观察者模式来处理。我们可以把**多个订阅者、客户称之为观察者**；需要同步给多个订阅者的数据封装到对象中，称之为**目标**。

观察者模式 Observer

- 核心：

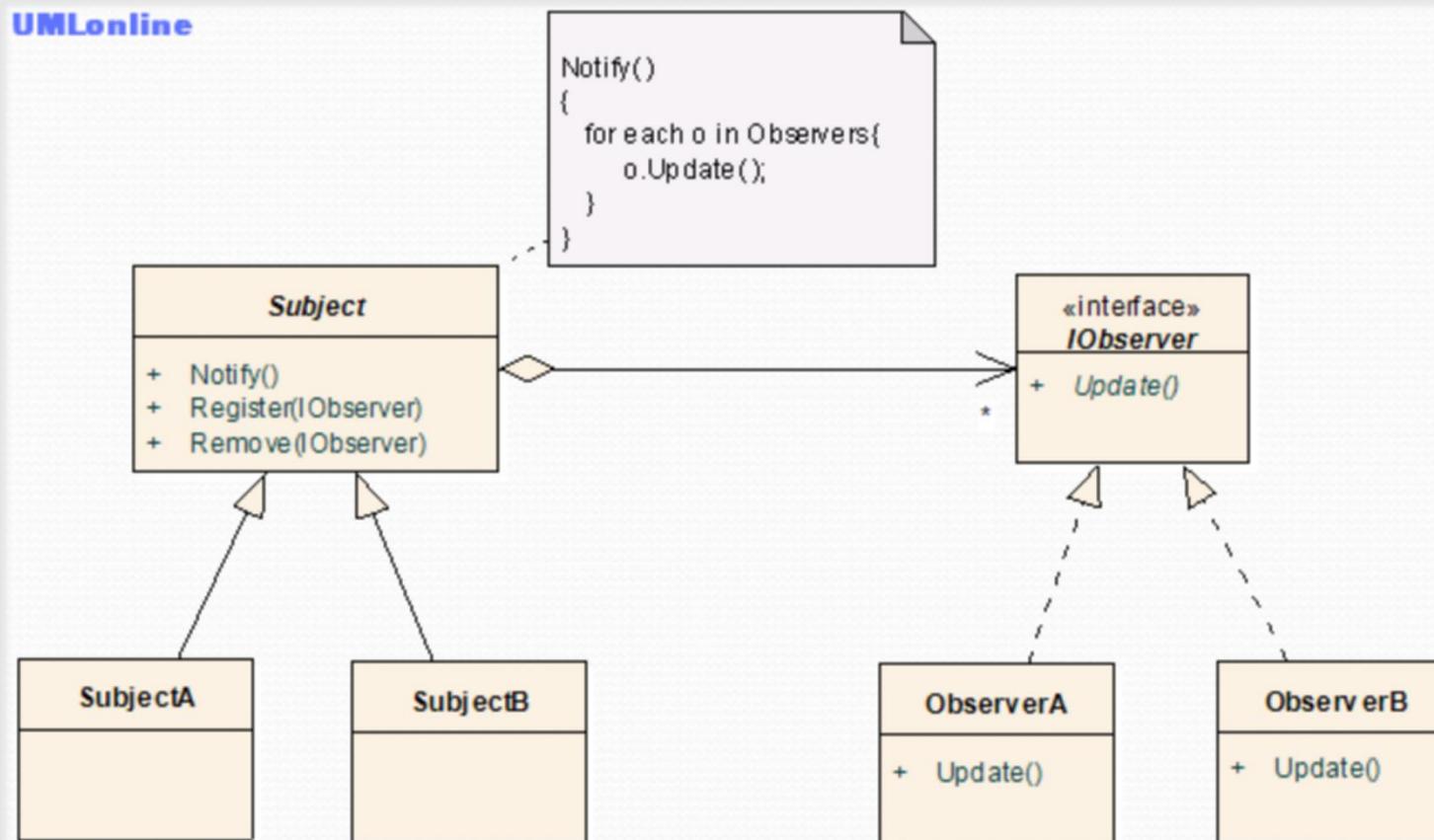
- 观察者模式主要用于1 : N的通知。当一个对象(**目标对象Subject或Observable**)的状态变化时，他需要及时告知一系列对象(**观察者对象,Observer**)，令他们做出响应
- 通知观察者的方式：
 - 推
 - 每次都会把通知以广播方式发送给所有观察者，所有观察者只能被动接收。
 - 拉
 - 观察者只要直到有情况即可。至于什么时候获取内容，获取什么内容，都可以自主决定。

消息发布

消息订阅

观察者模式 Observer

- UML类图



观察者模式 Observer

- JAVASE提供了**java.util.Observable**和**java.util.Observer**来实现观察者模式

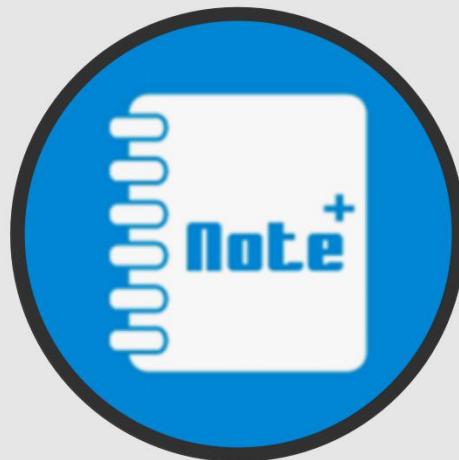
观察者模式 Observer

- 开发中常见的场景：
 - 聊天室程序的，服务器转发给所有客户端
 - 网络游戏(多人联机对战)场景中，服务器将客户端的状态进行分发
 - 邮件订阅
 - Servlet中，监听器的实现
 - Android中，广播机制
 - JDK的AWT中事件处理模型,基于观察者模式的委派事件模型(Delegation Event Model)
 - 事件源-----目标对象
 - 事件监听器-----观察者
 - 京东商城中，群发某商品打折信息

备忘录模式 memento

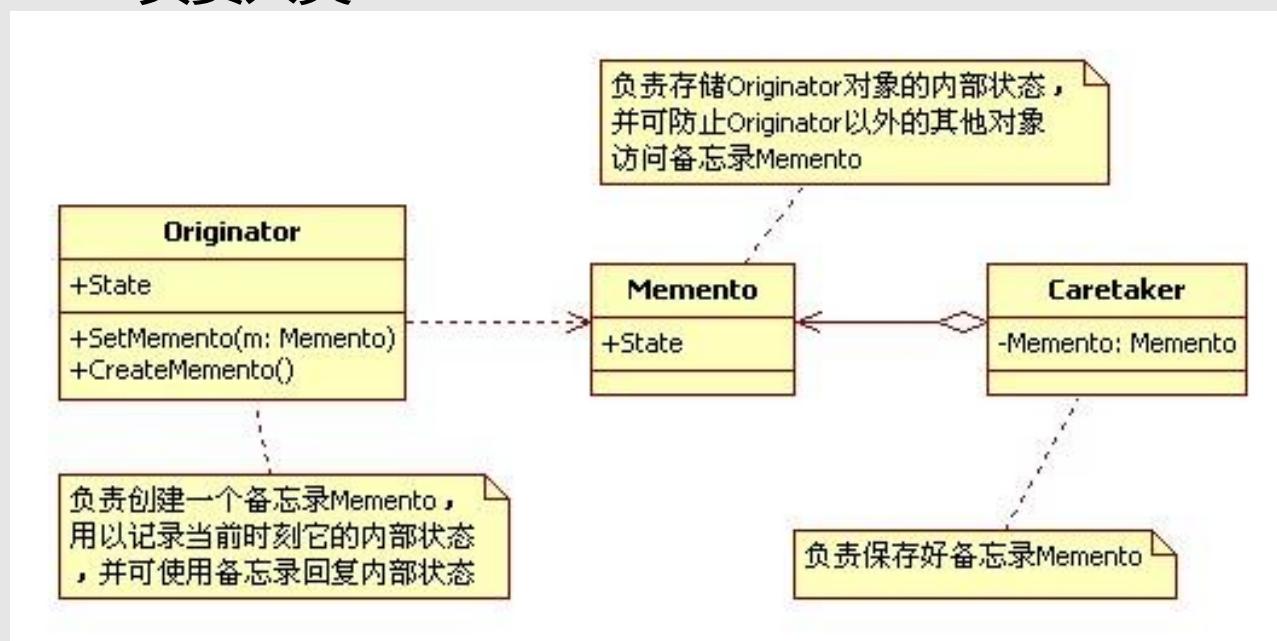
- 场景：

- 录入大批人员资料。正在录入当前人资料时，发现上一个人录错了，此时需要恢复上一个人的资料，再进行修改。
- Word文档编辑时，忽然电脑死机或断电，再打开时，可以看到word提示你恢复到以前的文档
- 管理系统中，公文撤回功能。公文发送出去后，想撤回来。



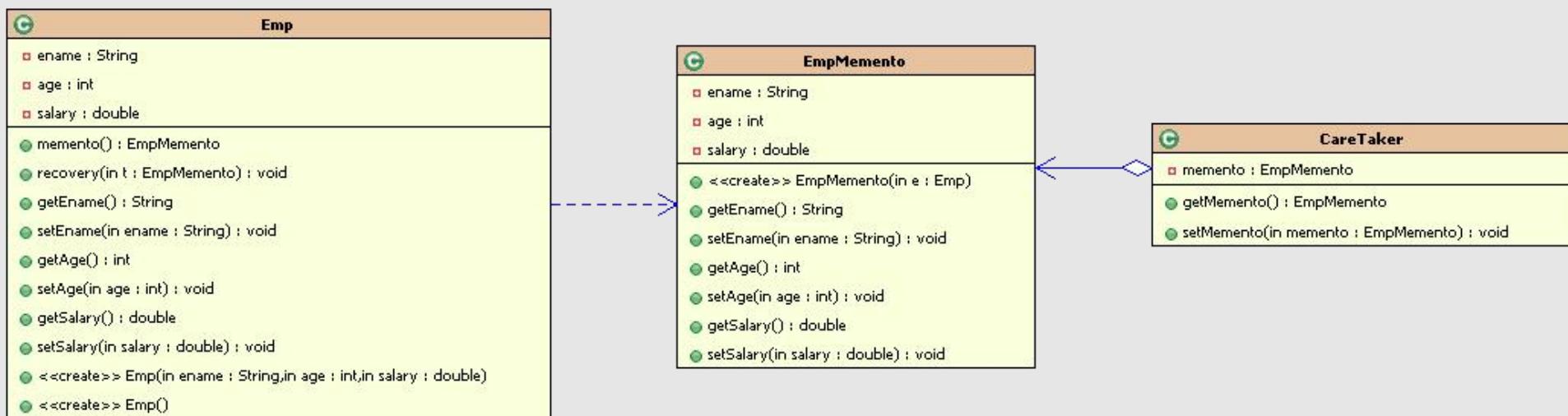
备忘录模式 memento

- 核心：
 - 就是保存某个对象内部状态的拷贝，这样以后就可以将该对象恢复到原先的状态。
- 结构：
 - 源发器类Originator
 - 备忘录类Memento
 - 负责人类Caretaker



备忘录模式 memento

- 课堂代码的类图



备忘录模式 memento

- 负责人类
 - 负责保存好的备忘录对象。
 - 可以通过增加容器，设置多个 “备忘点”

```
public class CareTaker {  
  
    private EmpMemento memento;  
  
    //  
    private List<EmpMemento> list = new ArrayList<EmpMemento>();  
  
    public EmpMemento getMemento() {  
        return memento;  
    }  
  
    public void setMemento(EmpMemento memento) {  
        this.memento = memento;  
    }  
}
```

可以通过容器设
置多个备份点！

备忘录模式 memento

- 备忘点较多时：

- 将备忘录压栈

```
public class CareTaker {  
    private Memento memento;  
    private Stack<Memento> stack = new Stack<Memento>();  
}
```

- 将多个备忘录对象，序列化和持久化

备忘录模式 memento

- **开发中常见的应用场景：**

- 棋类游戏中的，悔棋
- 普通软件中的，撤销操作
- 数据库软件中的，事务管理中的，回滚操作
- Photoshop软件中的，历史记录

设计模式汇总

- GOF23中设计模式一览表

创建型模式	结构型模式	行为型模式
简单工厂模式	适配器模式	责任链模式
抽象工厂模式	桥接模式	解释器模式
工厂方法模式	组合模式	模板方法模式
单例模式	装饰器模式	命令模式
建造者模式	外观模式	迭代器模式
原型模式	享元模式	中介者模式
	代理模式	备忘录模式
		观察者模式
		状态模式
		策略模式
		访问者模式

创建型模式的总结

- **创建型模式：都是用来帮助我们创建对象的！**
 - 单例模式
 - 保证一个类只有一个实例，并且提供一个访问该实例的全局访问点。
 - 工厂模式
 - 简单工厂模式
 - 用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
 - 工厂方法模式
 - 用来生产同一等级结构中的固定产品。（支持增加任意产品）
 - 抽象工厂模式
 - 用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）
 - 建造者模式
 - 分离了对象子组件的单独构造(由Builder来负责)和装配(由Director负责)。从而可以构造出复杂的对象。
 - 原型模式
 - 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式

结构型模式总结

• 结构型模式：关注对象和类的组织

代理模式	为真实对象提供一个代理，从而控制对真实对象的访问
适配模式	使原本由于接口不兼容不能一起工作的类可以一起工作
桥接模式	处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各个维度可以独立的扩展在抽象层建立关联。
组合模式	将对象组合成树状结构以表示”部分和整体”层次结构，使得客户可以统一的调用叶子对象和容器对象
装饰模式	动态地给一个对象添加额外的功能，比继承灵活
外观模式	为子系统提供统一的调用接口，使得子系统更加容易使用
享元模式	运用共享技术有效的实现管理大量细粒度对象，节省内存，提高效率

行为型模式总结

• 行为型模式汇总：

- 关注系统中对象之间的相互交互，研究系统在运行时对象之间的相互通信和协作，进一步明确对象的职责，共有11种模式。

职责链模式	避免请求发送者和接收者耦合，让多个对象都有可能接收请求，将这些对象连成一条链，并且沿着这条链传递请求，直到有对象处理为止
命令模式	将一个请求封装为一个对象，从而使得请求调用者和请求接收者解耦
解释器模式	描述如何为语言定义一个文法，如何解析
迭代器模式	提供了一种方法来访问聚合对象
中介者模式	通过一个中介对象来封装一系列的对象交互，使得各对象不需要相互引用
备忘录模式	捕获一个对象的内部状态，并保存之；需要时，可以恢复到保存的状态
观察者模式	当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新
状态模式	允许一个对象在其内部状态改变时改变它的行为
策略模式	定义一系列算法，并将每个算法封装在一个类中
模板方法	定义一个操作的算法骨架，将某些易变的步骤延迟到子类中实现
访问者模式	表示一个作用于某对象结构中的各元素的操作，它使得用户可以在不改变各元素的类的前提下定义作用于这些元素的新操作