



LANZHOU UNIVERSITY OF TECHNOLOGY

结课作业

题 目 设计模式期末测评作业

学生姓名 石 成 伟

学 号 18177004

专业班级 软件四班

学 院 软件学院

# 目录

[目录 2](#_Toc42678029)

[一．设计模式的分类 3](#_Toc42678030)

[二．各设计模式优缺点总结 3](#_Toc42678031)

[2.生成器模式（建造者模式） 4](#_Toc42678032)

[3.职责链模式 4](#_Toc42678033)

[4.蝇量模式（享元） 5](#_Toc42678034)

[5.解释器模式 5](#_Toc42678035)

[6.中介者模式 6](#_Toc42678036)

[7.备忘录模式 6](#_Toc42678037)

[8.原型模式 7](#_Toc42678038)

[9访问者模式 7](#_Toc42678039)

[10简单工厂模式 8](#_Toc42678040)

[11策略模式 8](#_Toc42678041)

[12装饰模式 9](#_Toc42678042)

[13代理模式 10](#_Toc42678043)

[14工厂方法模式 10](#_Toc42678044)

[15模板方法模式 18](#_Toc42678045)

[16外观模式 19](#_Toc42678046)

[17观察者模式 20](#_Toc42678047)

[18状态模式 20](#_Toc42678048)

[19适配器模式 21](#_Toc42678049)

[20组合模式 22](#_Toc42678050)

[21迭代器模式 22](#_Toc42678051)

[22单例模式 23](#_Toc42678052)

[23命令模式 24](#_Toc42678053)

# 一．设计模式的分类

## 总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

# 二．各设计模式优缺点总结

## 1.桥接模式

### ⑴优点：

① 将实现予以解耦，让它和界面之间不再永久绑定  
②抽象和实现可以独立扩展，不会影响到对方  
③对于“具体的抽象类”所做的改变，不会影响到客户。

### ⑵缺点：

① 增加了复杂度

### ⑶用途：

①适合使用在需要跨越多个平台的图形和窗口上  
② 当需要用不同的方式改变接口和实现时，你会发现桥接模式很好用。  
具体实例：跨平台的软件，不同电视机和不同的遥控器。

# 2.生成器模式（建造者模式）

## ⑴优点：

①将一个复杂对象的创建过程封装起来  
②允许对象通过多个步骤来创建，并且可以改变创建过程  
③向客户隐藏内部的表现  
④产品的实现可以被替换，因为客户只看到一个抽象的接口

## ⑵缺点：

① 与工厂模式相比，采用生成器模式创建对象更复杂，其客户，需要更多的知识领域。

## ⑶用处：

用来创建组合结构。

## ⑷典型例子：

画小人，构建小人分画头，画身体，画双手，黄双脚等不同构建部分，全部放在一起构建。

# 3.职责链模式

## ⑴优点：

①将请求的发送者和接收者解耦  
②可以简化你的对象，因为它不需要知道链的结构  
③通过改变链内的成员或调动他们的次序，允许你动态地新增或删除责

## ⑵缺点：

① 并不保证请求一定会被执行，如果没有任何对象处理它的话，它可能会落到链尾端之外  
②可能不容观察运行时的特征，有碍于除错。

## ⑶用途：

经常被使用在窗口系统中，处理鼠标和键盘之类的事件。  
当算法牵涉到一种链型运算，而且不希望处理过程中有过多的循环和条件选择语句，并  
且希望比较容易的扩充文法，可以采用职责链模式。  
①有多个对象处理请求，到底怎么处理在运行时确定。  
②希望在不明确指定接收者的情况下，向多个对象中的一个提交请求。  
③可处理一个请求的对象集合应该被动态指定。  
典型例子：  
一个请求发送给前台，前台表示我无权管理，将请求传递给财务部门，财务部门再……

# 4.蝇量模式（享元）

## ⑴优点：

①减少运行时对象实例的个数，节省内存  
②将许多“虚拟”对象的状态集中管理

## ⑵缺点：

一旦你实现了它，单个的逻辑实现将无法拥有独立而不同的行为

## ⑶用途：

当一个类有许多的实例，而这些实例能被同一方法控制的时候，我们就可以使用蝇量模式。  
具体场景：  
五子棋中的黑白子，改变坐标状态(x,y)，但用同一个实体。

# 5.解释器模式

## ⑴优点：

①将每一个语法规则表示成一个类，方便事先语言。  
②因为语法由许多类表示，所以你可以轻易地改变或扩展此语言  
③通过在类结构中加入新的方法，可以在解释的同时增加新的行为，例如打印格式的梅花或者进行复制的程序验证。

## ⑵缺点：

当语法规则数目太大时，这个模式可能会变得非常繁琐。

## ⑶用途：

①当你需要实现一个简答的语言时，使用解释器  
②当你有一个简单的语法，切简单比效率更重要时，使用解释器  
③可以处理脚本语言和编程语言  
典型例子：正则表达式

# 6.中介者模式

## ⑴优点：

①通过将对象彼此解耦，可以增加对象的复用性。  
②通过将控制逻辑集中，可以简化系统维护  
③可以让对象之间传递的消息变得简单而且大幅减少

## ⑵缺点：

①如果设计不当，中介者对象本身会变得过于复杂

## ⑶用途：

常常被用来协调相关的GUI组件(HF设计模式上的原话，这书附录A部分真的有点敷衍)经典例子：  
我租房，但没有户主信息，我和户主不能直接交替。没关系，中介者类有我和户主的信息，private我，private户主。而我和户主都认识中介者。我将信息传递给中介者，在我中调用中介者.获取信息（）方法，中介者获取信息后，再由中介者传递给户主。

# 7.备忘录模式

## ⑴优点：

①将被存储的状态放在外面，不要和关键对象混在一起，可以帮助维护内聚  
②保持关键对象的数据封装  
③提供了容易实现的恢复能力

## ⑵缺点：

①储存和恢复状态的过程可能相当耗时

## ⑶用途

备忘录模式用于存储状态，在java中可以使用序列化。  
经典例子：  
游戏中途保存游戏，这时候可以调用保存当前状态方法，再读取的时候调用读取。Java序列化机制在这方面非常的方便。

# 8.原型模式

## ⑴优点：

①向客户隐藏制造新实例的复杂性  
②提供让客户能够产生未知类型对象的选项  
③在某些环境下，复制对象比新建对象更有效

## ⑵缺点：

复制对象有时相当复杂

## ⑶用途：

在一个复制的类层次中，当系统必须从其中的许多类型创建新对象时，可以考虑原型模式。  
经典例子：  
随便拿一个类，给这个类写一个克隆方法，复制当前对象。或者直接用反序列化。

# 9访问者模式

## ⑴优点：

①允许你对组合结构加入新的操作，无需改变结构本身  
②想要加入新的操作相对容易  
③访问者所进行的操作，其代码是集中在一起的

## ⑵缺点：

①会打破组合类的封装  
②因为游走的功能牵涉其中，随意对组合结构的改变就更加困难。

## ⑶用途：

有比较稳定的数据结构，又有易于变化的算法的话，使用访问者模式就是比较合适的，因为访问者模式使得算法操作的增加变得容易。  
经典场景：特么访问者模式和翻译器模式，一个看不懂，一个怎么也不想看，到时候要是让我说这两个模式，我就自认倒霉。

# 10简单工厂模式

## 优点：

工厂类是整个模式的关键.包含了必要的逻辑判断,根据外界给定的信息,决定究竟应该创建哪个具体类的对象.通过使用工厂类,外界可以从直接创建具体产品对象的尴尬局面摆脱出来,仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的．明确了各自的职责和权利，有利于整个软件体系结构的优化。

## 缺点：

由于工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则，将全部创建逻辑集中到了一个工厂类中；它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。  
当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

## 用途：

工厂类负责创建的对象比较少；  
客户只知道传入工厂类的参数，对于如何创建对象（逻辑）不关心；  
由于简单工厂很容易违反高内聚责任分配原则，因此一般只在很简单的情况下应用。  
经典例子：这不是一个真正的设计模式

# 11策略模式

## ⑴优点：

①提供了一种替代继承的方法，而且保持了继承的优点，比继承更独立(算法独立，可以任意扩展)  
②避免程序使用多重条件转移语句，使系统更灵活，并易于扩展  
③遵守大部分常用设计原则，高内聚，低耦合

## ⑵缺点：

①每个具体策略类都会产生一个新类，所以会增加系统需要维护的类的数量。可以使用工厂方法来解决。

## 用途：

各个不同地区不同的纳税方法，HF中不同鸭子的方法。有多种鸭子，每个鸭子都有自己的行为，fly，quaak之类的。行为有行为类，继承同一接口实现不同操作，以此实现算法互换。

# 12装饰模式

## ⑴优点：

①装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。  
②通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。  
③有着比继承更加灵活的特性

## ⑵缺点：

由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

## ⑶用途：

当需要给一个类添加新的行为的时候，但基于开闭原则，就使用装饰模式。  
经典例子：  
 我穿衣服使用draw（）方法，在我穿好衣服后，我还打算再寄领带，而寄领带就是装饰类，我们可以把装饰类和对象（穿衣服类）继承于同一个接口，在装饰类的draw（）方法中调用super.draw（），然后再在这个方法里加上自己的特征。

# 13代理模式

## 优点：

向客户端隐藏了访问某个对象的细节及复杂性；可以动态地调用一个对象中的方法，且无需实现固定的接口。

## 缺点：

（个人见解切勿当真）总觉得代理者不够可靠，不能得到有效的保证，要是对象代理者在维护的时候，或者其他的做出了变动，对被代理的人来说可能带来损失。  
使用场景：  
①远程代理，可以隐藏一个对象存在于不同地址空间的事实  
②虚拟代理，比如html页面刷新的图片，图片一张嘴下载后才能看就是通过虚拟代理来替代了真实的图片，此时代理存储了真实图片的路径和尺寸  
③安全代理，用来控制真实对象的访问权限。一般用于对象应该有不同的访问权限的时候  
④智能指引，当调用真实的对象时，代理处理另外一些事。  
经典例子：  
 我玩wow，但又没有时间精力投入到里面，于是我请了个人来代练，代练的人和我都继承于玩家类。而代练者是认识我的，当代练的人开始刷副本的时候，调用代练者.刷副本（）方法,此时他在这个方法中实际调用的是我.刷副本。

# 14工厂方法模式

## ⑴优点：

①良好的封装性，代码结构清晰。一个对象创建是有条件约束的，如一个调用者需要一个具体的产品对象，只要知道这个产品的类名（或约束字符串）就可以了，不用知道创建对象的艰辛过程，减少模块间的耦合。  
②工厂方法模式的扩展性非常优秀。在增加产品类的情况下，只要适当地修改具体的工厂类或扩展一个工厂类，就可以完成“拥抱变化”。例如在我们的例子中，需要增加一个棕色人种，则只需要增加一个BrownHuman类，工厂类不用任何修改就可完成系统扩展。  
③屏蔽产品类。这一特点非常重要，产品类的实现如何变化，调用者都不需要关心，它只需要关心产品的接口，只要接口保持不表，系统中的上层模块就不要发生变化，因为产品类的实例化工作是由工厂类负责，一个产品对象具体由哪一个产品生成是由工厂类决定的。在数据库开发中，大家应该能够深刻体会到工厂方法模式的好处：如果使用JDBC连接数据库，数据库从MySql切换到Oracle，需要改动地方就是切换一下驱动名称（前提条件是SQL语句是标准语句），其他的都不需要修改，这是工厂方法模式灵活性的一个直接案例。  
④工厂方法模式是典型的解耦框架。高层模块值需要知道产品的抽象类，其他的实现类都不用关心，符合迪米特原则，我不需要的就不要去交流；也符合依赖倒转原则，只依赖产品类的抽象；当然也符合里氏替换原则，使用产品子类替换产品父类，没问题！

## 用途：

第一种情况是对于某个产品，调用者清楚地知道应该使用哪个具体工厂服务，实例化该具体工厂，生产出具体的产品来。Java Collection中的iterator() 方法即属于这种情况。  
 第二种情况，只是需要一种产品，而不想知道也不需要知道究竟是哪个工厂为生产的，即最终选用哪个具体工厂的决定权在生产者一方，它们根据当前系统的情况来实例化一个具体的工厂返回给使用者，而这个决策过程这对于使用者来说是透明的。  
典型例子：  
 车子继承vehicle（车）类，有小汽车卡，公交车bus等，车子工厂实现工厂接口，工厂接口有抽象方法vehicle produce vehicle（String type）方法，车子工厂中实现工厂方法vehicle  produce vehicle（String Type），方法中根据需要new新的车子。

## ⑷简单工厂模式

一个抽象产品类，派生出多个具体产品类

一个工厂类，通过逻辑判断，实例化所需的产品类

具体的描述

①抽象产品类：为各种具体产品声明抽象类或者接口

②具体产品类：抽象产品的具体实现，对应具体的产品

③工厂类：用户只关注产品，无需关注具体实现，生产出具体的产品

## ⑸工厂方法模式

一个抽象产品类，派生出多个具体产品类

一个抽象工厂类，派生出多个具体工厂类

每个具体工厂类只能生产出一个具体的产品类

具体的描述

①抽象产品类：为各种具体产品声明抽象类或者接口

②具体产品类：抽象产品的具体实现

③抽象工厂类：声明了一个方法，只对应一种产品

④具体工厂类：抽象工厂的实现，生成一个具体产品 (一组产品)

## ⑹抽象工厂模式

多个抽象产品类，每个抽象产品类都可以派生出多个具体产品类

一个抽象工厂类，可以派生出多个具体工厂类

每个具体工厂类都能生成出多个具体产品类 （后续都有相应代码，看了就会懂）

具体的描述

①抽象产品类：为各种具体产品声明抽象类或者接口

②具体产品类：抽象产品的具体实现

③抽象工厂类：声明了一组方法，每个方法对应一种类型产品

④具体工厂类：抽象工厂的实现，生成一组具体产品 (一组产品)

## ⑺代码实现

①简单工厂模式

抽象产品类

package com.nicecui.design.factory.simple;

/\*\*

\*

\* Fruits

\* @author NiceCui

\* @date 2018年1月23日上午11:33:37

\*

\*/

public abstract class Fruits {

public abstract void print();

}

具体产品类

// 苹果

public class Apple extends Fruits {

@Override

public void print() {

System.out.println("This is a Apple");

}

}

//橙子

public class Orange extends Fruits {

@Override

public void print() {

// TODO Auto-generated method stub

System.out.println("This is a Orange");

}

}

工厂类

package com.nicecui.design.factory.simple;

/\*\*

\*

\* FruitsFactory

\* @author NiceCui

\* @date 2018年1月23日上午11:33:20

\*

\*/

public class FruitsFactory {

public Fruits getFruit(String name) {

Fruits fruit = null;

if(""==name||name == null) {

return fruit;

}

switch (name) {

case "Apple":

fruit = new Apple();

break;

case "Orange":

fruit = new Orange();

break;

}

return fruit;

}

}

②工厂方法模式

实现一个加减的计算机

抽象产品类

package com.nicecui.design.factory.method;

public abstract class Operation {

public int A;

public int B;

public int getA() {

return A;

}

public void setA(int a) {

A = a;

}

public int getB() {

return B;

}

public void setB(int b) {

B = b;

}

public abstract void getResult() ;

}

具体产品类

//加法

public class OperationAdd extends Operation{

public int a;

public int b;

public OperationAdd() {

}

@Override

public void getResult() {

System.out.println("a+b="+(super.A+super.B));

}

}

// 减法

public class OperationSub extends Operation{

public int a;

public int b;

@Override

public void getResult() {

// TODO Auto-generated method stub

System.out.println("a-b="+(super.A - super.B));

}

}

抽象工厂类和具体工厂

package com.nicecui.design.factory.method;

public interface IFactory {

Operation createOperation();

}

class AddFactory implements IFactory{

@Override

public Operation createOperation() {

// TODO Auto-generated method stub

return new OperationAdd();

}

}

class SubFactory implements IFactory{

@Override

public Operation createOperation() {

// TODO Auto-generated method stub

return new OperationSub();

}

}

③抽象工厂模式

生产一辆汽车的工厂，配置不一样，有进口和本地发动机和在淘宝和京东购买的座椅

抽象产品类

//座椅

public interface Chair {

public void getChair();

}

//发动机

public interface Engine {

public void getEngine ();

}

具体产品类

/\*\*座椅\*\*/

//京东买的

public class JD\_Chair implements Chair{

@Override

public void getChair() {

// TODO Auto-generated method stub

System.out.println("从京东购买的座椅....");

}

}

//淘宝买的

public class TB\_Chair implements Chair {

@Override

public void getChair() {

// TODO Auto-generated method stub

System.out.println("从淘宝买的座椅.....");

}

}

/\*\*发动机\*\*/

//进口的

public class Import\_Engine implements Engine{

@Override

public void getEngine() {

// TODO Auto-generated method stub

System.out.println("使用进口发动机....");

}

}

//本地的

public class Local\_Engine implements Engine{

@Override

public void getEngine() {

// TODO Auto-generated method stub

System.out.println("使用本地发动机....");

}

}

抽象工厂

public interface CarFactory {

//一组方法 对应 一组产品 上面有讲过

public Chair getChair();

public Engine getEngine();

}

具体工厂

生产具体产品

//生产A型号的汽车

public class AcarFactory implements CarFactory{

@Override

public Chair getChair() {

// TODO Auto-generated method stub

return new JD\_Chair();

}

@Override

public Engine getEngine() {

// TODO Auto-generated method stub

return new Local\_Engine();

}

}

//生产B型号的汽车

public class BcarFactory implements CarFactory{

@Override

public Chair getChair() {

// TODO Auto-generated method stub

return new TB\_Chair();

}

@Override

public Engine getEngine() {

// TODO Auto-generated method stub

return new Import\_Engine();

}

}

## ◆简单工厂模式

## ⑴优点：

第一点当然是简单了，当耦合不是很大时候，简单工厂很适合最大的优点就是将类的实例化交给工厂类，工厂类中包含了必要的逻辑判断。客户端角度分析客户选择条件动态的实例化相关的类，去除了与具体产品的依赖。

## ⑵缺点：

简单工厂模式不利于拓展，违背了开放封闭原则，每次添加一个类，都要修改工厂类。

## ◆工厂方法模式

## ⑴优点：

工厂方法模式对简单工厂模式进行了升级，将类的实例化延迟到了其子类(也就是具体工厂类)。

拓展添加具体类，无需对抽象工厂类修改，只需要添加想要的具体类和具体工厂即可。

## ⑵缺点：

面对再复杂的有点力不从心，例如：如若增加另一个产品体系，刚才我们的工厂方法是实现计算机，若果我们要来一台电视机呢，还是得再去抽象工厂添加修改，这样也会违背开放封闭原则。

## ◆抽象工厂模式

## ⑴优点：

可以针对多个产品系，很容易组合产品系列，只需要去实现具体的工厂类即可；例如：我们再添加一个C型号的车 需要 进口发动机和京东座椅 那我们只需要去添加相应的具体工厂就可以了。

具有工厂方法模式解耦的优点。

产品系 进口发动机和本地发动机为两个不同的等级结构，搭载进口发动机的车子就是一个产品系。

## ⑵缺点：

对于产品系的扩展将非常费劲，例如我们要生产的所有车子，需要配备 进口轮胎或者本地轮胎，产品系中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。

# 15模板方法模式

## ⑴优点：

模板方法模式通过把不变的行为搬移到超类，去除子类中的重复代码来体现它的优势。模板方法模式提供了一个很好的代码复用平台。

## ⑵缺点：

没啥明显缺点吧？简单实用的设计模式。

## ⑶用途：

当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为会在子类中重复出现，我们可以通过模板方法牧师把这些行为搬移到单一的地方（方法）中，这样就帮助子类摆脱重复的不变行为的纠缠。  
典型例子：  
泡茶和泡咖啡两个动作很像，都有煮开水（）方法，放入茶或咖啡（）方法，把泡好的导入被子，加调料，这四个动作在一起算是完成泡咖啡行为。在这里，我们可以把泡茶和泡咖啡都继承于抽象类饮料中，而饮料类有个模板方法，templateMethod（）这个方法封装其他几个行为方法。而客户端想执行那几个方法的时候，只需要调用emplateMethod（）模板方法。我们会把templateMethod（）方法用final修饰，以免这个算法骨架被重写而改变。

# 16外观模式

## ⑴优点：

①对客户屏蔽了其子系统组件，因而减少了客户处理对象的数目，并使得子系统实用起来更方便  
②它实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。 Facade模式有助于建立层次结构系统，也有助于对对象之间的依赖关系分层。 Facade模式还可以消除复杂的循环依赖关系，这一点在客户程序与子系统是分别实现的时候尤为重要。  
③如果应用需要，它并不限制它们实用子系统类。因此你可以在系统易用性和通用性之间进行选择。

## ⑵缺点：

限制了客户的自由，减少了可变性。  
⑶用途：  
a) 为一个复杂子系统提供一个简单接口.  
b) 提高子系统的独立性.  
c) 在层次化结构中，可以使用Facade模式定义系统中每一层的入口。  
典型例子  
魔兽插件一键换装，会同时调用更换掉身上的装备接口和更换角色天赋的接口。

# 17观察者模式

## ⑴优点：

①观察者和被观察者之间是松耦合的，分别可以各自独立改变。

②Subject在发送广播通知的时候，无须指定具体的Observer，Observer可以自己决定是否要订阅Subject的通知。

③遵守大部分GRASP原则和常用设计原则，高内聚、低耦合。

## ⑵缺点：

①松耦合导致代码关系不明显，有时可能难以理解。

②如果一个对象被大量观察者订阅的话，在广播通知的时候可能会有效率问题。（毕竟只是简单的遍历）

## ⑶使用场景

①对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

②对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

经典例子：

看电视，天气预报

看新闻联播，观众们是观察者，新闻联播是对象，每个观察者都继承于一个统一的接口。而被观察的对象是新闻联播，新闻联播可以添加新的观察者，或者当移除观察者（关电视不看），新闻联播对象有个通知用户信息变更方法，当达到某种条件是，新闻联播对象会将信息都通知给观察者。

# 18状态模式

## ⑴优点：

①把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。

②将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。

③通过把各种状态转移逻辑分不到状态与其子类之间，来减少相互间的依赖。

## ⑵缺点：

@策略模式，每个具体状态类都会产生一个新类，所以会增加系统需要维护的类的数量。可以使用工厂方法来解决。（我只是把策略二字换成了状态）

## ⑶用途：

①一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。

②一个操作中含有庞大的多分支结构，并且这些分支决定于对象的状态。

经典例子：

我上班，上午点，下午，晚上各个时间段不同状态不同算法。这三个状态都继承一个统一的状态类。都有重写work（）方法，在我调用work的时候，会根据不同的状态调用不同的work。

# 19适配器模式

## ⑴优点：

①将目标类和适配者类解耦

②增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性,灵活性和扩展性都非常好，符合开闭原

## (2)缺点：

一次最多只能适配一个适配者类，而且目标抽象类只能为接口，不能为类，其使用有一定的局限性，不能将一个适配者类和他的子类同时适配到目标接口。

## ⑶用途：

想使用一个已经存在的类，但如果它的接口和你的要求不相同时。可以使用适配器模式。两个类所做的事情相同或相似，但接口不同时可以使用。

经典例子：

不同牌子电视机和不同牌子遥控器，里皮翻译用汉语布置恒大队员战术，不同牌子手机的软件。

球员都继承于中国人，都会说汉语，而里皮并不会说汉语。在不知战术的时候，我们需要给里皮一个翻译，这个翻译就是里皮的适配器。翻译可以和球员一样，也继承于中国人，也有说汉语这个方法。而翻译认识里皮，翻译在说汉语（）这个方法的时候，实际上是在说里面用意大利语的内容。

Fanyi.shuohanyu(){

Lipi.yidaliyu();

}

# 20组合模式

## ⑴优点：

①使客户端调用简单，客户端可以一致的使用组合结构或其中单个对象，用户就不必关心自己处理的是单个对象还是整个组合结构，这就简化了客户端代码。

②更容易在组合体内加入对象部件. 客户端不必因为加入了新的对象部件而更改代码。这一点符合开闭原则的要求，对系统的二次开发和功能扩展很有利！

## ⑵缺点：

组合模式不容易限制组合中的构件

## ⑶用途：

当你发现需求中是体现部分与整体层次的结构时，以及你希望可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑使用组合模式了。

典型例子：

树形结构，android界面控件

# 21迭代器模式

## ⑴优点：

①简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。

②可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。

③封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

## ⑵缺点：

对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐，大家可能都有感觉，像ArrayList，我们宁可愿意使用for循环和get方法来遍历集合，操作简易度完爆迭代。

## ⑶用途：

迭代器模式是与集合共生共死的，一般来说，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像java中的Collection，List、Set、Map等，这些集合都有自己的迭代器。假如我们要实现一个这样的新的容器，当然也需要引入迭代器模式，给我们的容器实现一个迭代器。

但是，由于容器与迭代器的关系太密切了，所以大多数语言在实现容器的时候都给提供了迭代器，并且这些语言提供的容器和迭代器在绝大多数情况下就可以满足我们的需要，所以现在需要我们自己去实践迭代器模式的场景还是比较少见的，我们只需要使用语言中已有的容器和迭代器就可以了。

# 22单例模式

## ⑴优点：

①实例控制，单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。

②灵活性，因为类控制了实例化过程，所以类可以灵活更改实例化过程。

## ⑵缺点：

①可能的开发混淆，使用单例对象（尤其在类库中定义的对象）时，开发人员必须记住自己不能使用new 关键字实例化对象。因为可能无法访问库源代码，因此应用程序开发人员可能会意外发现自己无法直接实例化此类。

②不能解决删除单个对象的问题。在提供内存管理的语言中（例如基于.NET Framework的语言），只有单例类能够导致实例被取消分配，因为它包含对该实例的私有引用。在某些语言中（如 C++），其他类可以删除对象实例，但这样会导致单例类中出现悬浮引用。

## ⑶用途：

系统只需要一个实例的对象，而这个对象又会被经常创建。  
典型例子：

数据库类里的Model，例如User，表单中显示用户信息会经常使用User.Update之类的方法，此时使用单例模式可以节省效率。

# 23命令模式

## ⑴优点：

①能比较容易的设计一个命令队列

②可以较容易的将命令加入日志

③允许接收请求的一方是否处理请求

④可以容易的实现对请求的添加和删除

⑤加进新的具体命令类不会影响到其他的类，增加具体命令类很容易

⑥把请求一个操作的对象与指导怎么执行一个操作对象分开（解耦）

## ⑵缺点：

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

## ⑶用途：

①系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。

②系统需要在不同的时间指定请求、将请求排队和执行请求。

③系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。

④系统需要将一组操作组合在一起，即支持宏命令。

典型例子：

饭店点菜，每个菜都是命令类的子类，每次点菜都是一道命令（鱼香肉丝命令1，肉末茄子命令2），服务员是中间类，将命令存储在一个数组中（ArrayList），再将命令提交给厨师来处理。服务员存储的数组命令可以很方便查询记录。