华南农业大学信息学院

软件体系结构综合性实验报告

李志彬

201527010413

2018年7月完稿

目 录

目录

[模式一：简单工厂模式](#_Toc20172_WPSOffice_Level1) [5](#_Toc20172_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc14195_WPSOffice_Level1) [5](#_Toc14195_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc16856_WPSOffice_Level1) [5](#_Toc16856_WPSOffice_Level1)

[(3) 具体待解决的问题](#_Toc27404_WPSOffice_Level1) [5](#_Toc27404_WPSOffice_Level1)

[(4) UML设计](#_Toc22515_WPSOffice_Level1) [5](#_Toc22515_WPSOffice_Level1)

[(5) 核心代码](#_Toc21162_WPSOffice_Level1) [5](#_Toc21162_WPSOffice_Level1)

[(6) 实现效果](#_Toc28022_WPSOffice_Level1) [6](#_Toc28022_WPSOffice_Level1)

[(7) 模式的优点](#_Toc20138_WPSOffice_Level1) [6](#_Toc20138_WPSOffice_Level1)

[(8) 模式的不足](#_Toc24304_WPSOffice_Level1) [6](#_Toc24304_WPSOffice_Level1)

[模式二：工厂方法模式](#_Toc30828_WPSOffice_Level1) [7](#_Toc30828_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc23357_WPSOffice_Level1) [7](#_Toc23357_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc14951_WPSOffice_Level1) [7](#_Toc14951_WPSOffice_Level1)

[(4) 具体待解决的问题](#_Toc1479_WPSOffice_Level1) [7](#_Toc1479_WPSOffice_Level1)

[(5) UML设计](#_Toc10756_WPSOffice_Level1) [7](#_Toc10756_WPSOffice_Level1)

[(5) 核心代码](#_Toc10083_WPSOffice_Level1) [8](#_Toc10083_WPSOffice_Level1)

[(6) 实现效果](#_Toc14704_WPSOffice_Level1) [8](#_Toc14704_WPSOffice_Level1)

[(7) 模式的优点](#_Toc19322_WPSOffice_Level1) [8](#_Toc19322_WPSOffice_Level1)

[(8) 模式的不足](#_Toc18615_WPSOffice_Level1) [9](#_Toc18615_WPSOffice_Level1)

[模式三：抽象工厂模式](#_Toc21285_WPSOffice_Level1) [9](#_Toc21285_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc6366_WPSOffice_Level1) [9](#_Toc6366_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc5720_WPSOffice_Level1) [9](#_Toc5720_WPSOffice_Level1)

[(4) UML设计](#_Toc30955_WPSOffice_Level1) [9](#_Toc30955_WPSOffice_Level1)

[(5) 核心代码](#_Toc11102_WPSOffice_Level1) [9](#_Toc11102_WPSOffice_Level1)

[(6) 实现效果](#_Toc1008_WPSOffice_Level1) [12](#_Toc1008_WPSOffice_Level1)

[(8) 模式的不足](#_Toc28460_WPSOffice_Level1) [12](#_Toc28460_WPSOffice_Level1)

[模式九：类适配器模式](#_Toc16527_WPSOffice_Level1) [13](#_Toc16527_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc32087_WPSOffice_Level1) [13](#_Toc32087_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc13720_WPSOffice_Level1) [13](#_Toc13720_WPSOffice_Level1)

[(3) 具体待解决的问题](#_Toc31924_WPSOffice_Level1) [13](#_Toc31924_WPSOffice_Level1)

[(4) UML设计](#_Toc25782_WPSOffice_Level1) [13](#_Toc25782_WPSOffice_Level1)

[(5) 核心代码](#_Toc3553_WPSOffice_Level1) [13](#_Toc3553_WPSOffice_Level1)

[(6) 实现效果](#_Toc25402_WPSOffice_Level1) [16](#_Toc25402_WPSOffice_Level1)

[(8) 模式的不足](#_Toc21332_WPSOffice_Level1) [16](#_Toc21332_WPSOffice_Level1)

[模式五：对象适配器模式](#_Toc19077_WPSOffice_Level1) [17](#_Toc19077_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc1697_WPSOffice_Level1) [17](#_Toc1697_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc17584_WPSOffice_Level1) [17](#_Toc17584_WPSOffice_Level1)

[(3) 具体待解决的问题](#_Toc27916_WPSOffice_Level1) [17](#_Toc27916_WPSOffice_Level1)

[(4) UML设计](#_Toc24400_WPSOffice_Level1) [17](#_Toc24400_WPSOffice_Level1)

[(5) 核心代码](#_Toc19884_WPSOffice_Level1) [17](#_Toc19884_WPSOffice_Level1)

[(6) 实现效果](#_Toc23953_WPSOffice_Level1) [19](#_Toc23953_WPSOffice_Level1)

[(8) 模式的不足](#_Toc23976_WPSOffice_Level1) [19](#_Toc23976_WPSOffice_Level1)

[模式六：单例模式](#_Toc19456_WPSOffice_Level1) [20](#_Toc19456_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc31572_WPSOffice_Level1) [20](#_Toc31572_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc27571_WPSOffice_Level1) [20](#_Toc27571_WPSOffice_Level1)

[(2) 具体待解决的问题](#_Toc5942_WPSOffice_Level1) [20](#_Toc5942_WPSOffice_Level1)

[(4) UML设计](#_Toc30648_WPSOffice_Level1) [20](#_Toc30648_WPSOffice_Level1)

[(3) 核心代码](#_Toc25153_WPSOffice_Level1) [20](#_Toc25153_WPSOffice_Level1)

[(6) 实现效果](#_Toc7918_WPSOffice_Level1) [21](#_Toc7918_WPSOffice_Level1)

[(7) 模式的优点](#_Toc21181_WPSOffice_Level1) [21](#_Toc21181_WPSOffice_Level1)

[(8) 模式的不足](#_Toc7608_WPSOffice_Level1) [21](#_Toc7608_WPSOffice_Level1)

[模式七：多例模式](#_Toc24003_WPSOffice_Level1) [23](#_Toc24003_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc32291_WPSOffice_Level1) [23](#_Toc32291_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc28972_WPSOffice_Level1) [23](#_Toc28972_WPSOffice_Level1)

[(2) 具体待解决的问题](#_Toc31361_WPSOffice_Level1) [23](#_Toc31361_WPSOffice_Level1)

[(4) UML设计](#_Toc23142_WPSOffice_Level1) [23](#_Toc23142_WPSOffice_Level1)

[(5) 核心代码](#_Toc7549_WPSOffice_Level1) [23](#_Toc7549_WPSOffice_Level1)

[(6) 实现效果](#_Toc26769_WPSOffice_Level1) [24](#_Toc26769_WPSOffice_Level1)

[(8) 模式的不足](#_Toc21561_WPSOffice_Level1) [24](#_Toc21561_WPSOffice_Level1)

[模式八：安全组合模式](#_Toc19655_WPSOffice_Level1) [25](#_Toc19655_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc2551_WPSOffice_Level1) [25](#_Toc2551_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc4491_WPSOffice_Level1) [25](#_Toc4491_WPSOffice_Level1)

[(3) 具体待解决的问题](#_Toc25170_WPSOffice_Level1) [25](#_Toc25170_WPSOffice_Level1)

[(4) UML设计](#_Toc2727_WPSOffice_Level1) [25](#_Toc2727_WPSOffice_Level1)

[(5) 核心代码](#_Toc3615_WPSOffice_Level1) [25](#_Toc3615_WPSOffice_Level1)

[(6) 实现效果](#_Toc26882_WPSOffice_Level1) [26](#_Toc26882_WPSOffice_Level1)

[(7) 模式的不足](#_Toc22631_WPSOffice_Level1) [27](#_Toc22631_WPSOffice_Level1)

[模式九：一致性组合模式](#_Toc14910_WPSOffice_Level1) [27](#_Toc14910_WPSOffice_Level1)

[(1) 解决问题类型](#_Toc13337_WPSOffice_Level1) [27](#_Toc13337_WPSOffice_Level1)

[(2) 解决设计思路](#_Toc6677_WPSOffice_Level1) [27](#_Toc6677_WPSOffice_Level1)

[(3) 具体待解决的问题](#_Toc6246_WPSOffice_Level1) [27](#_Toc6246_WPSOffice_Level1)

[(4) UML设计](#_Toc19409_WPSOffice_Level1) [27](#_Toc19409_WPSOffice_Level1)

[(5) 核心代码](#_Toc10470_WPSOffice_Level1) [27](#_Toc10470_WPSOffice_Level1)

[(6) 实现效果](#_Toc25684_WPSOffice_Level1) [30](#_Toc25684_WPSOffice_Level1)

[(8) 模式的不足](#_Toc20468_WPSOffice_Level1) [30](#_Toc20468_WPSOffice_Level1)

[综合性实验的心得体会](#_Toc12679_WPSOffice_Level1) [31](#_Toc12679_WPSOffice_Level1)

## 模式一：简单工厂模式

### 解决问题类型

简单工厂模式适用情况包括：工厂类负责创建的对象比较少；客户端只知道传入工厂类的参数，对于如何创建对象不关心。

### 解决设计思路

**又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。**

### 具体待解决的问题

当客户需要什么产品时，通过简单工厂来获得。

### (4) UML设计

### (5) 核心代码

**package** cn.edu.scau.cmi.lizhibin.simplefactory;

**import** cn.edu.scau.cmi.lizhibin.domain.Circle;

**import** cn.edu.scau.cmi.lizhibin.domain.Rectangle;

**import** cn.edu.scau.cmi.lizhibin.domain.Shape;

**public** **class** SimpleFactory {

**public** **static** Shape getShape(String s) {

**if**(s.equals("Rectangle")) {

**return** **new** Rectangle();

}

**else** {

**return** **new** Circle();

}

}

}

### 实现效果





### (7) 模式的优点

### 简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责

### (8) 模式的不足

## 工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

## 模式二：工厂方法模式

### (1) 解决问题类型

### 客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定。

### (2) 解决设计思路

### **在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象。**

### 具体待解决的问题

对于每个新的产品类需要额外增加一个新的工厂类，增加了额外开销。

### UML设计



### (5) 核心代码

**package** cn.edu.scau.cmi.lzb.factory;

**import** cn.edu.scau.cmi.lizhibin.domain.\*;

**public** **class** CircleFactory **implements** Factory{

@Override

**public** Shape getShape() {

// **TODO** Auto-generated method stub

**return** **new** Circle();

}

}

**package** cn.edu.scau.cmi.lzb.factory;

**import** cn.edu.scau.cmi.lizhibin.domain.\*;

**public** **interface** Factory {

**public** Shape getShape();

}

**package** cn.edu.scau.cmi.lzb.factory;

**import** cn.edu.scau.cmi.lizhibin.domain.\*;

**public** **class** RectangleFactory **implements** Factory{

@Override

**public** Shape getShape() {

// **TODO** Auto-generated method stub

**return** **new** Rectangle();

}

}

### (6) 实现效果

### 

### 

### (7) 模式的优点

### 工厂方法模式的主要优点是增加新的产品类时无须修改现有系统，并封装了产品对象的创建细节，系统具有良好的灵活性和可扩展性

### (8) 模式的不足

## 其缺点在于增加新产品的同时需要增加新的工厂，导致系统类的个数成对增加，在一定程度上增加了系统的复杂性。

## 模式三：抽象工厂模式

### (1) 解决问题类型

### 系统中有多于一个的产品族，而每次只使用其中某一产品族；属于同一个产品族的产品将在一起使用。

### (2) 解决设计思路

### 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。(3) 具体待解决的问题

### 增加新的产品等级结构很复杂，需要修改抽象工厂和所有的具体工厂类

### (4) UML设计

### (5) 核心代码

**public** **abstract** **class** AbstractFactory {

**public** **static** AbstractFactory getFactory(String name) {

**switch**(name) {

**case** "Apple" :**return** **new** AppleFactory();

**case** "Huawei" :**return** **new** HuaweiFactory();

**case** "Xiaomi" :**return** **new** XiaomiFactory();

}

**return** **null**;

}

**public** **abstract** Phone createPhone() ;

**public** **abstract** Computer createComputer();

**public** **abstract** Video createVideo() ;

}

**public** **class** AppleFactory **extends** AbstractFactory{

**public** Phone createPhone() {

**return** **new** ApplePhone();

}

**public** Computer createComputer() {

**return** **new** AppleComputer();

}

**public** Video createVideo() {

**return** **new** AppleVideo();

}

}

**public** **class** HuaweiFactory **extends** AbstractFactory{

**public** Phone createPhone() {

**return** **new** HuaweiPhone();

}

**public** Computer createComputer() {

**return** **new** HuaweiComputer();

}

**public** Video createVideo() {

**return** **new** HuaweiVideo();

}

}

**public** **class** HuaweiFactory **extends** AbstractFactory{

**public** Phone createPhone() {

**return** **new** HuaweiPhone();

}

**public** Computer createComputer() {

**return** **new** HuaweiComputer();

}

**public** Video createVideo() {

**return** **new** HuaweiVideo();

}

}

### (6) 实现效果

### 

### (7) 模式的优点

### 隔离了具体类的生成，使得客户并不需要知道什么被创建，而且每次可以通过具体工厂类创建一个产品族中的多个对象，增加或者替换产品族比较方便，增加新的具体工厂和产品族很方便。

### (8) 模式的不足

## 主要缺点在于增加新的产品等级结构很复杂，需要修改抽象工厂和所有的具体工厂类，对“开闭原则”的支持呈现倾斜性。

## 模式九：类适配器模式

### 解决问题类型

将一个类的接口转换成客户希望的另外一个接口。A d a p t e r 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一起工作。

### (2) 解决设计思路

 Adapter 类既继承了 Adaptee （被适配类），也实现了 Target 接口（因为 Java 不支持多继承，所以这样来实现），在 Client 类中我们可以根据需要选择并创建任一种符合需求的子类，来实现具体功能。

### (3) 具体待解决的问题

### 将交流电压220v通过适配器适配成直流5v来满足客户接口。

### (4) UML设计



### (5) 核心代码

**package** cn.edu.scau.cmi.lizhibin.Adapter;

**public** **class** Adaptee {//存在220V的电压

**public** **int** Output\_220() {

//System.out.println("交流电压220V");

**int** value = 220;

**return** value;

}

}

**package** cn.edu.scau.cmi.lizhibin.Adapter;

**public** **class** AdapterOfClass **extends** Adaptee **implements** Target{//采用类适配器去适配适合客户的接口

**public** **int** Output\_5() {

// **TODO** Auto-generated method stub

**int** V = Output\_220();

System.***out***.println("适配器开始工作");

**int** v = V /44;

System.***out***.println("将交流220V转变为直流5V");

**return** v;

}

}

**package** cn.edu.scau.cmi.lizhibin.Adapter;

**public** **interface** Target {//客户需要5V的接口

**public** **int** Output\_5();

}

Client:

**package** cn.edu.scau.cmi.lizhibin.client;

**import** cn.edu.scau.cmi.lizhibin.Adapter.\*;

**public** **class** AdapterClient {

**public** AdapterClient(Target target) {

System.***out***.println("客户开始充电");

**if**(target **instanceof** AdapterOfClass) {

System.***out***.println("类适配器使用");

**int** v = target.Output\_5();

System.***out***.println("充电电压为" + v +"V");

}

**else** {

System.***out***.println("对象适配器使用");

**int** v = target.Output\_5();

System.***out***.println("充电电压为" + v +"V");

}

}

**public** **static** **void** main(String[] args) {

AdapterClient client1 = **new** AdapterClient(**new** AdapterOfClass());

AdapterClient client2 = **new** AdapterClient(**new** AdapterOfObject(**new** Adaptee()));

System.***out***.println("结束");

}

}

### (6) 实现效果

### (7) 模式的优点

### 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。

### (8) 模式的不足

每次有不同的adaptee需要添加额外的适配器类，增加了额外开销。

## 模式五：对象适配器模式

### 解决问题类型

将一个类的接口转换成客户希望的另外一个接口。A d a p t e r 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一起工作。

### (2) 解决设计思路

### 适配器实现我们的目标接口，但是并不继承需要被适配的类。而是通过在适配器的构造函数中将需要被适配的类传递进来从而进行适配。

### (3) 具体待解决的问题

### 将交流电压220v通过适配器适配成直流5v来满足客户接口。

### (4) UML设计

### 

### (5) 核心代码

package cn.edu.scau.cmi.lizhibin.Adapter;

public class Adaptee {//存在220V的电压

public int Output\_220() {

//System.out.println("交流电压220V");

int value = 220;

return value;

}

}

package cn.edu.scau.cmi.lizhibin.Adapter;

public class AdapterOfObject implements Target{

Adaptee adaptee;

public AdapterOfObject(Adaptee adaptee) {

this.adaptee = adaptee;

}

public int Output\_5() {

// TODO Auto-generated method stub

int V = adaptee.Output\_220();

System.out.println("适配器开始工作");

int v = V / 44;

System.out.println("将交流电压220V转变为5V");

return v;

}

}

**package** cn.edu.scau.cmi.lizhibin.Adapter;

**public** **interface** Target {//客户需要5V的接口

**public** **int** Output\_5();

}

### (6) 实现效果

### 

### (7) 模式的优点

### 把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和他的子类都适配到目标接口。

### (8) 模式的不足

与类适配器模式相比，要想置换适配者类的方法就不容易。

## 模式六：单例模式

### (1) 解决问题类型

系统只需要一个实例对象；客户调用类的单个实例只允许使用一个公共访问点。

### (2) 解决设计思路

### **自行实例化并向整个系统提供这个实例，提供全局访问**

### 具体待解决的问题

通过单例模式来实现创建主席的过程，由于主席只能有一个，所以第二次创建时依然还是原来的主席。

### (4) UML设计

### 

### 核心代码

**package** cn.edu.scau.cmi.lizhibin.singletonpattern;

**public** **class** Persident {

**private** String name;

**private** **static** Persident *instance*;

**private** Persident(String name) {

**this**.name = name;

}

**public** **static** Persident getInstance(String \_name) {

**if** (*instance* == **null** && \_name != **null** && !\_name.equals("")) {

*instance* = **new** Persident(\_name);

}

**return** *instance*;

}

}

### (6) 实现效果

### 

### 

### (7) 模式的优点

### 1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例      2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。      3.提供了对唯一实例的受控访问。      4.由于在系统内存中只存在一个对象，因此可以 节约系统资源，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。

### (8) 模式的不足

1.不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。   
  2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。   
    3.单例类的职责过重，在一定程度上违背了“单一职责原则”。   
    4.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的单例类，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；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

## 模式七：多例模式

### 解决问题类型

### 当一个系统需要确定若干数目的同一类对象时，需要采用多例模式，多例模式，是为了防止并发问题；即一个请求改变了对象的状态，此时对象又处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处理；

### (2) 解决设计思路

### 对类的构造函数私有化，提供一个全局创建方法，并且在类中添加一个数据结构，来存储不同的对象。

### 具体待解决的问题

该系统存在确定数量的“元帅”，所以需要采取多例模式来保证这个系统的元帅数量。

### (4) UML设计

### 

### (5) 核心代码

**package** cn.edu.scau.cmi.lizhibin.singletonpattern;

**import** java.util.\*;

**public** **class** Marshal {

**private** **static** Set<Marshal> *marshals* =**new** HashSet<Marshal>();

**private** String name;

**private** Marshal(String name) {

**this**.name = name;

}

**private** String getName() {

**return** name;

}

**public** **static** **void** getInstance(String name) {

**if**(name != **null** && name.equals("") && *marshals*.size() < 3) {

*marshals*.add(**new** Marshal(name));

}

}

**public** **static** String[] getMarshals() {

String[] m1 = **new** String[3];

**int** i = 0;

**for**(Marshal marshal:*marshals*) {

String s = marshal.getName();

m1[i] = s;

i++;

}

**return** m1;

}

### }

### (6) 实现效果

### (7) 模式的优点

1. 多例类可以有多个实例  
   2. 多例类必须自己创建自己的实例，并管理自己的实例，和向外界提供自己的实例

3 解决在并发问题下，请求使得对象的状态发生变化时，能够再次新建一个实例来处理请求。

### (8) 模式的不足

跟单例模式一样，难于扩展和维护。

## 模式八：安全组合模式

### (1) 解决问题类型

### **组合多个对象形成树形结构以表示“整体-部分”的结构层次。组合模式对单个对象（即叶子对象）和组合对象（即容器对象）的使用具有一致性。**

### (2) 解决设计思路

### 对于被实现的接口，接口的抽象方法应该使得接口类都能够实现，而个别接口类可以额外添加接口里面没有的方法。

### (3) 具体待解决的问题

### 一个organization里面有member和group，group又可以包含organization。

### (4) UML设计

### 

### (5) 核心代码

**package** cn.edu.scau.cmi.lizhibin.componsite.safe;

**public** **abstract** **class** Member {

}

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Organization **extends** Member{

**private** List<Member> team = **new** ArrayList<Member>();

**public** String add(Member member) {

team.add(member);

System.***out***.println("添加成功");

**return** "添加成功";

}

**public** String delete(Member member) {

team.remove(member);

System.***out***.println("删除成功");

**return** "删除成功";

}

}

### (6) 实现效果

### 

### (7) 模式的优点

**安全组合模式中，在抽象构件Component中没有声明任何用于管理成员对象的方法，而是在Composite类中声明并实现这些方法。**这种做法是安全的，因为根本不向叶子对象提供这些管理成员对象的方法，对于叶子对象，客户端不可能调用到这些方法。

### 模式的不足

**安全组合模式的缺点是不够透明**，因为叶子构件和容器构件具有不同的方法，且容器构件中那些用于管理成员对象的方法没有在抽象构件类中定义，因此客户端不能完全针对抽象编程，**必须有区别地对待叶子构件和容器构件**。

## 模式九：一致性组合模式

### (1) 解决问题类型

**组合多个对象形成树形结构以表示“整体-部分”的结构层次。组合模式对单个对象（即叶子对象）和组合对象（即容器对象）的使用具有一致性。**

### (2) 解决设计思路

### 接口提供了大量的管理方法，对于一些实现类来说，这些方法并不能实现，故对不能实现的方法采用异常处理来解决。

### (3) 具体待解决的问题

### 一个organization里面有member和group，group又可以包含organization。

### (4) UML设计

### 

### (5) 核心代码

|  |
| --- |
| **package** cn.edu.scau.cmi.lizhibin.componsite.transparent;  **import** java.util.List;  **public** **class** Organization **implements** Member{  **private** List<Member> team;  **public** **void** add(Member member) {  team.add(member);  System.***out***.println("添加成功");    }  **public** **void** delete(Member member) {    team.remove(member);  System.***out***.println("删除成功");  }  } |

|  |
| --- |
| **package** cn.edu.scau.cmi.lizhibin.client;  **import** cn.edu.scau.cmi.lizhibin.componsite.transparent.\*;  **public** **class** CompositeTransparentClient {  **public** **static** **void** main(String[] args) {  Member group1 = **new** Organization();    Member group2 = **new** Organization();  Member people = **new** Person();  group1.add(group2);  group1.add(people);    }  } |

|  |
| --- |
| **package** cn.edu.scau.cmi.lizhibin.componsite.transparent;  **public** **interface** Member {  **public** **void** add(Member member);  **public** **void** delete(Member member);  } |

|  |
| --- |
| **package** cn.edu.scau.cmi.lizhibin.componsite.transparent;  **public** **class** Person **implements** Member{    @Override  @Deprecated  **public** **void** ~~add~~(Member member) **throws** UnsupportedOperationException{  **throw** **new** UnsupportedOperationException();  }    @Override  @Deprecated  **public** **void** ~~delete~~(Member member) **throws** UnsupportedOperationException{  **throw** **new** UnsupportedOperationException();  }  } |

### (6) 实现效果

### 

### (7) 模式的优点

### **确保所有的构件类都有相同的接口**。在客户端看来，叶子对象与容器对象所提供的**方法是一致的**，客户端可以相同地对待所有的对象。

### (8) 模式的不足

透明组合模式的**缺点是不够安全**，因为叶子对象和容器对象在本质上是有区别的。叶子对象不可能有下一个层次的对象，即不可能包含成员对象，因此为其提供add()、remove()以及getChild()等方法是没有意义的，这在编译阶段不会出错，但在运行阶段如果调用这些方法可能会出错（如果没有提供相应的错误处理代码）。

## 综合性实验的心得体会

这次的综合性实验，是对我之前所学的软件体系结构知识的一次全面总结和回顾，通过这一次的综合性试验，我从中学到了很多。首先，对于软件体系的各种设计模式，包括工厂模式，单例模式，适配器模式，组合模式等，加深了对他们的理解和掌握，能够认识到他们的适用场景以及优缺点，其次，这次的综合性试验也学到了一些新的技术，总之，这次的综合性试验我受益匪浅。