设计模式课程实验报告（项目一）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称** | 设计模式 | **班级** | 20软件2 | **实验日期** | 第3-5周周三第五六节 |
| **姓名** | 徐小龙 | **学号** | B20200306207 | **指导老师** | 张肖霞 |
| **实验评分** |  | | | | |
| **教师评语** |  | | | | |
| **实验名称** | 创建型模式 | | | | |
| 实验目的及要求 | 以本实验指导中给定的实验模式为实验实例，掌握简单工厂模式、工厂方法模式、抽象工厂模式、原型模式和单例模式的应用。 | | | | |
|
|
|
|
|
| 实验环境 | 1、硬件平台：计算机CPU 奔腾Ⅲ；内存64兆以上（最好128兆以上）。  2、软件：Windows 2000 server；Eclipse。 | | | | |
| 实验步骤 | 1. 掌握各个模式的定义； 2. 理解各个模式的结构图；   3、使用相关的工具绘制给定背景的UML类图；  4、依据类图使用合适的面向对象语言实现；  5、分析和总结各模式的优缺点。 | | | | |
| 实验内容及实验结果（截图） | 实验内容：（以下内容至少完成四道题，其中完成四道题及格，五道题中等，六道题良好，七道题优秀）  1、在某绘图软件中提供了多种大小不同的画笔（Pen），并且可以给画笔制定不同的颜色，某设计人员针对画笔的结构设计了如图所示的初始类图。  LBV8H~_[)C8A4UFLRNP$@SH  试分析该设计存在的不足，并对该设计方案进行重构，使得增加新的大小的笔和增加新的颜色都比较方便。  UML图：    实现代码：    BlackColor：  package test1\_1;  public class BlackColor implements Color{  @Override  public String getColor() {  return "BlackColor";  }  }  Color：  package test1\_1;  public interface Color {  public String getColor();  }  LargeSize：  package test1\_1;  public class LargeSize implements Size {  public String getSize() {  return "LargeSize";  }  }  Pen：  package test1\_1;  public class Pen {  Color color;  Size size;    public Pen(Color color, Size size) {  super();  this.color = color;  this.size = size;  }  public void draw() {  System.out.println(size.getSize()+"画"+color.getColor());  }  }  RedColor：  package test1\_1;  public class RedColor implements Color {  public String getColor() {  return "RedColor";  }  }  Size：  package test1\_1;  public interface Size {  public String getSize();  }  SmallSize：  package test1\_1;  public class SmallSize implements Size {  public String getSize() {  return "SmallSize";  }  }  test:  package test1\_1;  public class test {  public static void main(String []args) {  Pen pen=new Pen(new RedColor(),new LargeSize());  pen.draw();  pen=new Pen(new BlackColor(),new SmallSize());  pen.draw();  }  }  测试结果：    总结：  方便后续添加其他的大小和颜色  2、请模拟驾驶员开车的应用场景，驾驶员有驾驶的行为，汽车有移动的行为，请用面向对象的方法模拟驾驶员驾驶各类汽车的场景（本场景中至少要模拟宝马、奔驰、大众三类汽车），通过设计理解面向对象的多态特型、理解类之间的关系和面向对象的设计原则。请分析你的设计中是否运用了面向对象的多态特型，遵循了面向对象的哪些设计原则。  要求使用相关的工具先绘制UML类图，再依据类图的设计编写程序实现。  UML类图：    程序代码：    Benz.java  package test1\_2;  public class Benz extends Car{  public String driving() {  return "Benz";  }  }  BWM.java  package test1\_2;  public class BWM extends Car{  public String driving() {  return "BWM";  }    }  Car.java  package test1\_2;  public abstract class Car {  public abstract String driving();  }  Daz.java  package test1\_2;  public class Daz extends Car{  public String driving() {  return "Daz";  }  }  Drive.java  package test1\_2;  public class Drive {  String name;    public Drive(String name) {  super();  this.name = name;  }  public void driving(Car car) {  System.out.println(name+"正在开"+car.driving());  }  }  test.java  package test1\_2;  public class test {  public static void main(String []args) {  Drive dr=new Drive("张三");  dr.driving(new BWM());  }  }    总结：  利用多态能使得只需要更改很少一部分代码就能选择开其他的车  3、通过仔细试运用面向使用简单工厂模式设计能够实现包含加法（+）、减法（-）、乘法（\*）、除法（/）四种运算的计算机程序，要求输入两个数和运算符，得到运算结果。要求使用相关的工具绘制UML类图并严格按照类图的设计编写程序实现。  UML图：    代码：    Add.java  package test1\_3;  public class Add extends Arthmetic {  public float getResult(float numA, float numB) {  return numA+numB;  }    }  Arthmetic.java  package test1\_3;  public abstract class Arthmetic {  public abstract float getResult(float numA,float numB);  }  ArthmeticFactory.java  package test1\_3;  public class ArthmeticFactory {  public Arthmetic getMethod(String str) {  if(str.equals("+")) {  return new Add();  }  else if(str.equals("-")) {  return new Del();  }  else if(str.equals("\*")) {  return new Mul();  }  else if(str.equals("/")) {  return new Div();  }  else {  System.out.println("请输入正确的符号");  return null;  }    }  public static void main(String []args) {    }  }  Client.java  package test1\_3;  import java.util.Scanner;  public class Client {  public static void main(String []args) {  Scanner sc=new Scanner(System.in);  float numA=0,numB=0,ans=0;  String c=null;  System.out.println("请输入一个数");  try {  if(sc.hasNext()) {  numA=sc.nextFloat();  }  } catch (Exception e) {  System.out.println("请确认输入的是数字");  sc.close();  return;  }  System.out.println("请输入+ - \* /中的一个符号");  if(sc.hasNext()) {  c=sc.next();  }  if(!c.equals("-")&&!c.equals("\*")&&!c.equals("+")&&!c.equals("/")) {  System.out.println("请确认输入的是+ - \* /中的一个符号");  sc.close();  return;  }  System.out.println("请输入一个数");  try {  if(sc.hasNext()) {  numB=sc.nextFloat();  }  } catch (Exception e) {  System.out.println("请确认输入的是数字");  sc.close();  return;  }  sc.close();  if(c.equals("/")&&numB==0) {  System.out.println("被除数不能为0");  return;  }  ArthmeticFactory artfy=new ArthmeticFactory();  ans=artfy.getMethod(c).getResult(numA, numB);  System.out.println(ans);  }  }  Del.java  package test1\_3;  public class Del extends Arthmetic{  public float getResult(float numA, float numB) {  return numA-numB;  }    }  Div.java  package test1\_3;  public class Div extends Arthmetic{  @Override  public float getResult(float numA, float numB) {  return numA/numB;  }    }  Mul.java  package test1\_3;  public class Mul extends Arthmetic{  @Override  public float getResult(float numA, float numB) {  return numA\*numB;  }  }  运行截图：  参考运行结果如下：  第一种情形：    第二种情形：    第三种情形：    总结：  使用工厂，使得创建都在工厂中完成，减少了Cilent的代码量  4、使用工厂方法模式设计能够实现包含加法（+）、减法（-）、乘法（\*）、除法（/）四种运算的计算机程序，要求输入两个数和运算符，得到运算结果。要求使用相关的工具绘制UML类图并严格按照类图的设计编写程序实现。  UML类图：    代码：    Add.java  package test1\_4;  public class Add extends Arthmetic {  public float getResult(float numA, float numB) {  return numA+numB;  }    }  AddFactory.java  package test1\_4;  public class AddFactory extends Factory{  public Arthmetic getMethod(String str) {  if(str.equals("+"))  return new Add();  else return null;  }  }  Arthmetic.java  package test1\_4;  public abstract class Arthmetic {  public abstract float getResult(float numA,float numB);  }  Client.java  package test1\_4;  import java.util.Scanner;  public class Client {  public static void main(String []args) {  Scanner sc=new Scanner(System.in);  float numA=0,numB=0,ans=0;  String c=null;  System.out.println("请输入一个数");  try {  if(sc.hasNext()) {  numA=sc.nextFloat();  }  } catch (Exception e) {  System.out.println("请确认输入的是数字");  sc.close();  return;  }  System.out.println("请输入+ - \* /中的一个符号");  if(sc.hasNext()) {  c=sc.next();  }  if(!c.equals("-")&&!c.equals("\*")&&!c.equals("+")&&!c.equals("/")) {  System.out.println("请确认输入的是+ - \* /中的一个符号");  sc.close();  return;  }  System.out.println("请输入一个数");  try {  if(sc.hasNext()) {  numB=sc.nextFloat();  }  } catch (Exception e) {  System.out.println("请确认输入的是数字");  sc.close();  return;  }  sc.close();  if(c.equals("/")&&numB==0) {  System.out.println("被除数不能为0");  return;  }  Factory fay;  if(c.equals("+")) {  fay=new AddFactory();  }  else if(c.equals("-")) {  fay=new DelFactory();  }  else if(c.equals("\*")) {  fay=new MulFactory();  }  else {  fay=new DivFactory();  }  ans=fay.getMethod(c).getResult(numA, numB);  System.out.println(ans);  }  }  Del.java  package test1\_4;  public class Del extends Arthmetic{  public float getResult(float numA, float numB) {  return numA-numB;  }    }  DelFactory.java  package test1\_4;  public class DelFactory extends Factory{  public Arthmetic getMethod(String str) {  if(str.equals("-"))  return new Del();  else return null;  }  }  Div.java  package test1\_4;  public class Div extends Arthmetic{  @Override  public float getResult(float numA, float numB) {  return numA/numB;  }    }  DivFactory.java  package test1\_4;  public class DivFactory extends Factory{  public Arthmetic getMethod(String str) {  if(str.equals("/"))  return new Div();  else return null;  }  }  Factory.java  package test1\_4;  public abstract class Factory {  public abstract Arthmetic getMethod(String str);  }  Mul.java  package test1\_4;  public class Mul extends Arthmetic{  @Override  public float getResult(float numA, float numB) {  return numA\*numB;  }  }  MulFactory.java  package test1\_4;  public class MulFactory extends Factory{  public Arthmetic getMethod(String str) {  if(str.equals("\*"))  return new Mul();  else return null;  }  }  实现：  第一种情况：    第二种情况：    第三个情况：    总结：  使用抽象工厂，能有效减少后期由于过多的内容添加使得工厂过于臃肿的问题  5、一个电器工厂可以生产多种类型的电器，如海尔工厂可以生产海尔电视机、海尔空调等，TCL工厂可以生产TCL电视机，TCL空调等，相同品牌的电器构成一个产品族，而相同类型的电器构成了一个产品等级结构，现使用抽象工厂模拟该场景。要求使用相关的工具绘制UML类图并严格按照类图的设计编写程序实现。  UML图：    代码：    Air.java:  package test1\_5;  public abstract class Air {  protected String name;  public void work() {  System.out.println(name+" is woring");  }  }  Client.java:  package test1\_5;  public class Client {  public static void main(String args[]) {  Factory fac;  fac=new HairFactory();  TV tvH=fac.creatTV();  tvH.play();  System.out.println("------------------");  fac=new TCLFactory();  Air airT=fac.creatAir();  airT.work();  }  }  Factory.java:  package test1\_5;  public interface Factory {  public TV creatTV();  public Air creatAir();  }  HairFactory.java:  package test1\_5;  public class HairFactory implements Factory{  public TV creatTV() {  return new TCLTV("HairTV");  }  public Air creatAir() {  return new TCLAir("HairAir");  }  }  TCLAir.java:  package test1\_5;  public class TCLAir extends Air{  public TCLAir(String name) {  this.name=name;  }  }  TCLFactory.java:  package test1\_5;  public class TCLFactory implements Factory{  public TV creatTV() {  return new TCLTV("TCLTV");  }  public Air creatAir() {  return new TCLAir("TCLAir");  }  }  TCLTV.java:  package test1\_5;  public class TCLTV extends TV{  public TCLTV(String name) {  this.name=name;  }  }  TV.java:  package test1\_5;  public abstract class TV {  protected String name;  public void play() {  System.out.println(name+" is playing");  }  }  测试：    总结：  使用抽象工厂，能有效减少后期由于过多的内容添加使得工厂过于臃肿的问题   1. 请为某销售管理系统设计并实现一个客户类（Customer），在客户类中包含客户姓名，年龄，客户地址等信息，其中客户地址的类型为Address引用类型，客户地址中包含地址名称，邮政编码等信息。请用浅克隆和深克隆分别实现Customer对象的复制并比较这两种克隆方式的异同。要求使用相关的工具绘制UML类图并严格按照类图的设计编写程序实现。   第一个：  UML图：    代码：    Address.java:  package test1\_6\_1;  public class Address {  String name;  String post;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getPost() {  return post;  }  public void setPost(String post) {  this.post = post;  }    }  Client.java:  package test1\_6\_1;  public class Client {  public static void main(String []args) {  Customer cusold,cusnew;  cusold=new Customer();  cusold.setName("张三");  cusold.setAge(18);  Address add=new Address();  add.setName("开福区");  add.setPost("1111");  cusold.setAddress(add);  cusnew=cusold.clone();  System.out.println("原对象的基本信息如下：");  System.out.println("客户姓名："+cusold.getName()+  "，年龄是"+cusold.getAge()+"，地址是"+cusold.getAddress().getName()+  "，邮编是"+cusold.getAddress().getPost());  System.out.println("浅克隆对象的基本信息如下：");  System.out.println("客户姓名："+cusnew.getName()+  "，年龄是"+cusnew.getAge()+"，地址是"+cusnew.getAddress().getName()+  "，邮编是"+cusnew.getAddress().getPost());  System.out.println("-------修改原对象姓名和地址信息，分别显示原对象和浅克隆对象的信息-------");  cusold.setName("李四");  cusold.getAddress().setName("长沙大学");  System.out.println("原对象的基本信息如下：");  System.out.println("客户姓名："+cusold.getName()+  "，年龄是"+cusold.getAge()+"，地址是"+cusold.getAddress().getName()+  "，邮编是"+cusold.getAddress().getPost());  System.out.println("浅克隆对象的基本信息如下：");  System.out.println("客户姓名："+cusnew.getName()+  "，年龄是"+cusnew.getAge()+"，地址是"+cusnew.getAddress().getName()+  "，邮编是"+cusnew.getAddress().getPost());  System.out.println("判断原对象和浅克隆对象是否是同一个引用对象？"+(cusold==cusnew));  System.out.println("判断原对象和浅克隆对象的地址是否是同一个引用对象？"+(cusold.getAddress()==cusnew.getAddress()));  }  }  Customer.java:  package test1\_6\_1;  public class Customer implements Cloneable{  String name;  int age;  Address address;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public Address getAddress() {  return address;  }  public void setAddress(Address address) {  this.address = address;  }  public Customer clone() {  Object obj=null;  try {  obj=super.clone();  return (Customer)obj;  } catch (CloneNotSupportedException e) {  System.out.println("克隆失败");  return null;  }  }  }  测试：    第二个情况； UML图：    代码：    Address.java:  package test1\_6\_2;  import java.io.Serializable;  public class Address implements Serializable {  String name;  String post;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getPost() {  return post;  }  public void setPost(String post) {  this.post = post;  }  }  Client.java:  package test1\_6\_2;  public class Client {  public static void main(String []args) {  Customer cusold,cusnew=null;  cusold=new Customer();  cusold.setName("张三");  cusold.setAge(18);  Address add=new Address();  add.setName("开福区");  add.setPost("1111");  cusold.setAddress(add);  try {  cusnew=cusold.deepClone();  } catch (Exception e) {  System.out.println("克隆失败");  }  System.out.println("原对象的基本信息如下：");  System.out.println("客户姓名："+cusold.getName()+  "，年龄是"+cusold.getAge()+"，地址是"+cusold.getAddress().getName()+  "，邮编是"+cusold.getAddress().getPost());  System.out.println("深克隆对象的基本信息如下：");  System.out.println("客户姓名："+cusnew.getName()+  "，年龄是"+cusnew.getAge()+"，地址是"+cusnew.getAddress().getName()+  "，邮编是"+cusnew.getAddress().getPost());  System.out.println("-------修改原对象姓名和地址信息，分别显示原对象和深克隆对象的信息-------");  cusold.setName("李四");  cusold.getAddress().setName("长沙大学");  System.out.println("原对象的基本信息如下：");  System.out.println("客户姓名："+cusold.getName()+  "，年龄是"+cusold.getAge()+"，地址是"+cusold.getAddress().getName()+  "，邮编是"+cusold.getAddress().getPost());  System.out.println("深克隆对象的基本信息如下：");  System.out.println("客户姓名："+cusnew.getName()+  "，年龄是"+cusnew.getAge()+"，地址是"+cusnew.getAddress().getName()+  "，邮编是"+cusnew.getAddress().getPost());  System.out.println("判断原对象和浅克隆对象是否是同一个引用对象？"+(cusold==cusnew));  System.out.println("判断原对象和浅克隆对象的地址是否是同一个引用对象？"+(cusold.getAddress()==cusnew.getAddress()));  }  }  Customer.java:  package test1\_6\_2;  import java.io.ByteArrayInputStream;  import java.io.ByteArrayOutputStream;  import java.io.IOException;  import java.io.ObjectInputStream;  import java.io.ObjectOutputStream;  import java.io.OptionalDataException;  import java.io.Serializable;  public class Customer implements Serializable{  String name;  int age;  Address address;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public Address getAddress() {  return address;  }  public void setAddress(Address address) {  this.address = address;  }  public Customer deepClone() throws IOException,ClassNotFoundException,OptionalDataException{  ByteArrayOutputStream bao =new ByteArrayOutputStream();  ObjectOutputStream oos =new ObjectOutputStream(bao);  oos.writeObject(this);    ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteArray());  ObjectInputStream ois =new ObjectInputStream(bis);  return (Customer)ois.readObject();  }  }  测试：    总结：  浅克隆是会在克隆的时候直接把引用复制上去，而深克隆是重新创建一个，所以在更改深克隆的地址不会出现浅克隆的一起呗更改的情况  7、使用单例模式设计一个多文档窗口（注：在Java AWT/Swing开发中可使用JDesktopPane和JInternalFrame来实现），要求在主窗体中某个内部子窗体只能实例化一次，即只能弹出一个相同的子窗体。要求使用相关的工具绘制UML类图并严格按照类图的设计编写程序实现。 | | | | |
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
| 小结 | 通过这一次实验我受益匪浅，了解到了不同的模式区别以及优劣，也了解到了深克隆与浅克隆之间的区别，让我对设计模式的了解更进一步 | | | | |
|
|
|
|  |  | | | | |