设计模式课程实验报告（项目二）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称** | 设计模式 | **班级** | 20软件2 | **实验日期** | 第6-9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** | 结构型设计模式 | | | | |
| 实验目的及要求 | 掌握适配器模式、桥接模式，外观模式，装饰模式和代理模式的的定义，结构及在特定环境下的应用。 | | | | |
|
|
|
|
|
| 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） | 按要求完成以下各题   1. 现有一个接口DataOperation定义了排序方法sort(int[])和查找方法search(int[],int)，已知类QuickSort的quickSort(int[])方法实现了快速排序算法，类BinarySearch的binarySearch(int[],int)方法实现了二分查找算法。现使用适配器模式设计一个系统，在不修改源代码的情况下将类QuickSort和类BinarySearch的方法适配到DataOperation接口中。绘制类图并编程实现。   结构：    Adapter.java:  package test2\_1;  public class Adapter implements DataOperation{  public void sort(int[] a) {  QuickSort qu=new QuickSort();  qu.quickSort(a);  }  public int search(int[] a, int q) {  BinarySearch bi=new BinarySearch();  return bi.binarySearch(a, q);  }    }  BinarySearch.java:  package test2\_1;  public class BinarySearch {  public int binarySearch(int []a,int key) {  return java.util.Arrays.binarySearch(a, key);  }  }  Client.java:  package test2\_1;  import java.util.Scanner;  public class Client {  public static void main(String[] args) {  int []a={6,5,7,2,3,1};  System.out.println("原数组为：");  Client ci=new Client();  ci.printf(a);  System.out.println("快速排序后数组的信息为：");  Adapter ad=new Adapter();  ad.sort(a);  ci.printf(a);  System.out.println("请输入你要查找的数：");  Scanner sc=new Scanner(System.in);  int t=-1;  try {  if(sc.hasNext()) {  t=sc.nextInt();  }  } catch (Exception e) {    }  int ans=ad.search(a, t);  if(ans<0)System.out.println("没有这个数");  else System.out.println("数字"+t+"的索引为"+ans);    }  public void printf(int[] a) {  for(int x:a) {  System.out.print(x+" ");  }  System.out.println();  }  }  DataOperation.java:  package test2\_1;  public interface DataOperation {  public void sort(int [] a);  public int search(int []a,int q);  }  QuickSort.java:  package test2\_1;  public class QuickSort {  public void quickSort(int []a) {  java.util.Arrays.sort(a);  }  }  类图：    测试：    二、在HuntBird游戏中，需要模拟不同鸟的多种行为，鸟类中提供了对其各种行为信息进行显示的show()方法，目前游戏需要呈现的是鸽子和企鹅的飞和游泳行为，现使用桥接模式设计该游戏（设计需要考虑到后期鸟的种类和行为的变化性），要求绘制该应用的UML类图，并实现。（能力较好的同学可以综合运用工厂模式和桥接模式来完成，其中工厂模式用于创建各种鸟类）  参考实现效果如下：  结构：  代码： Bird.java:  package test2\_2;  public abstract class Bird {  protected Swim s;  protected Fly f;  public void setS(Swim s) {  this.s = s;  }  public void setF(Fly f) {  this.f = f;  }  public abstract void show();  }  CanFly.java:  package test2\_2;  public class CanFly implements Fly{  public void showFly()  {  System.out.print("能飞");  }  }  CanSwim.java:  package test2\_2;  public class CanSwim implements Swim{  public void showSwim()  {  System.out.print("能游");  }  }  Client.java:  package test2\_2;  import java.util.Scanner;  public class Client {  public static void main(String []args) {  Scanner sc=new Scanner(System.in);  while(true) {  System.out.println("请输入要测试的鸟，1表示鸽子，2表示老鹰，3表示企鹅，其他表示退出");  int op=sc.nextInt();  Bird bird;  if(op==1)bird=new GeZi();  else if(op==2)bird=new LaoYing();  else if(op==3)bird=new QiE();  else break;  bird.show();  }  sc.close();  }  }  Fly.java:  package test2\_2;  public interface Fly {  public void showFly();  }  GeZi.java:  package test2\_2;  public class GeZi extends Bird{  public void show() {  setF(new CanFly());  setS(new NoSwim());  System.out.print("鸽子");  s.showSwim();  System.out.print(",");  f.showFly();  System.out.println("");  }  }  LaoYing.java:  package test2\_2;  public class LaoYing extends Bird{  public void show() {  setF(new CanFly());  setS(new NoSwim());  System.out.print("老鹰");  s.showSwim();  System.out.print(",");  f.showFly();  System.out.println("");  }  }  NoFly.java:  package test2\_2;  public class NoFly implements Fly{  public void showFly()  {  System.out.print("不能飞");  }  }  NoSwim.java:  package test2\_2;  public class NoSwim implements Swim{  public void showSwim()  {  System.out.print("不能游");  }  }  QiE.java:  package test2\_2;  public class QiE extends Bird{  public void show() {  setF(new NoFly());  setS(new CanSwim());  System.out.print("企鹅");  s.showSwim();  System.out.print(",");  f.showFly();  System.out.println("");  }  }  Swim.java:  package test2\_2;  public interface Swim {  public void showSwim();  }  类图：  测试：    三（可二选一，全部完成可加分）  （3.1）在电脑主机（MainFrame）中只需要按下主机的开机按钮（on（））,即可调用其它硬件设备和软件的启动方法，如内存（Memory）的自检（check()）、CPU的运行（run()）、硬盘（HardDisk）的读取（read()）、操作系统（OS）的载入（load())等，如果某一过程发生错误则电脑启动失败。使用外观模式模拟该过程，绘制类图并使用Java语言模拟实现。  （3.2）某系统需要提供一个文件加密模块，加密流程包括三个操作，分别是读取源文件、加密、保存加密之后的文件。读取文件和保存文件使用流来实现，这三个操作相对独立，其业务代码封装在三个不同的类中。现在需要提供一个统一的加密外观类，用户可以直接使用该加密外观类完成文件的读取、加密和保存三个操作，而不需要与每一个类交互，使用外观模式设计该加密模块，要求编程模拟实现。  结构：    代码：  AbstractEncryptFacade.java:  package test2\_3\_2;  public abstract class AbstractEncryptFacade {  public abstract void fileEncrypt(String fileNameSrc,String fileNameDes);  }  AbstructCipherMachine.java:  package test2\_3\_2;  public abstract class AbstructCipherMachine {  public abstract String encrypt(String plainText);  }  CipherMachine.java:  package test2\_3\_2;  public class CipherMachine extends AbstructCipherMachine{  public String encrypt(String plainText) {  System.out.println("数据加密，将明文转换为密文");  String es="";  for(int i=0;i<plainText.length();i++) {  String c=String.valueOf(plainText.charAt(i)%7);  es+=c;  }  System.out.println(es);  return es;  }  }  Client.java:  package test2\_3\_2;  public class Client {  public static void main(String []args) {  AbstractEncryptFacade ef=new EncryptFacade();  ef.fileEncrypt("C:\\Users\\28013\\Desktop\\编程\\JAVA\\设计模式\\bin\\test2\_3\_2\\src.txt","C:\\Users\\28013\\Desktop\\编程\\JAVA\\设计模式\\bin\\test2\_3\_2\\des.txt");  ef=new NewEncryptFacade();  ef.fileEncrypt("C:\\Users\\28013\\Desktop\\编程\\JAVA\\设计模式\\bin\\test2\_3\_2\\src.txt","C:\\Users\\28013\\Desktop\\编程\\JAVA\\设计模式\\bin\\test2\_3\_2\\des.txt");  }  }  des.txt:  EncryptFacade.java:  package test2\_3\_2;  public class EncryptFacade extends AbstractEncryptFacade{  private FileReader reader;  private AbstructCipherMachine cipher;  private FileWriter writer;  public EncryptFacade() {  reader=new FileReader();  cipher=new CipherMachine();  writer=new FileWriter();  }  public void fileEncrypt(String filrNameSrc,String fileNameDes) {  String plainStr=reader.read(filrNameSrc);  String encryptStr=cipher.encrypt(plainStr);  writer.writer(encryptStr, fileNameDes);  }  }  FileReader.java:  package test2\_3\_2;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  public class FileReader {  public String read(String fileNameSrc) {  System.out.println("读取文件，获取明文：");  StringBuffer sb=new StringBuffer();  try {  FileInputStream inFS=new FileInputStream(fileNameSrc);  int data;  while((data=inFS.read())!=-1) {  sb=sb.append((char)data);  }  inFS.close();  System.out.println(sb.toString());  } catch (FileNotFoundException e) {  System.out.println("文件不存在！");  }  catch (IOException e) {  System.out.println("文件操作错误！");  }  return sb.toString();  }  }  FileWriter.java:  package test2\_3\_2;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  public class FileWriter {  public void writer(String encryptStr,String fileNameDes) {  System.out.println("保存密文，写入文件。");  try {  FileOutputStream outFS=new FileOutputStream(fileNameDes);  outFS.write(encryptStr.getBytes());  outFS.close();  } catch (FileNotFoundException e) {  System.out.println("文件不存在！");  }  catch (IOException e) {  System.out.println("文件操作错误！");  }  }  }  NewCipherMachine.java:  package test2\_3\_2;  public class NewCipherMachine extends AbstructCipherMachine{  public String encrypt(String plainText) {  System.out.println("数据加密，将明文转换为密文");  String es="";  for(int i=0;i<plainText.length();i++) {  String c=String.valueOf(plainText.charAt(i)%4);  es+=c;  }  System.out.println(es);  return es;  }  }  NewEncryptFacade.java:  package test2\_3\_2;  public class NewEncryptFacade extends AbstractEncryptFacade{  private FileReader reader;  private AbstructCipherMachine cipher;  private FileWriter writer;  public NewEncryptFacade() {  reader=new FileReader();  cipher=new NewCipherMachine();  writer=new FileWriter();  }  public void fileEncrypt(String filrNameSrc,String fileNameDes) {  String plainStr=reader.read(filrNameSrc);  String encryptStr=cipher.encrypt(plainStr);  writer.writer(encryptStr, fileNameDes);  }  }  类图：  测试：    四（可二选一，若全部完成可加分）  （4.1）某家咖啡店在卖咖啡时可以根据顾客的要求在其中加入各种配料，咖啡店会根据所加入的配料来计算总费用，咖啡店所供应的咖啡机配料的种类和价格如下表所示：   |  |  |  |  | | --- | --- | --- | --- | | 咖啡 | 价格/杯(元) | 配料 | 价格/份(元) | | 蓝山咖啡（BlueCoffee） | 8 | 牛奶（Milk） | 2 | | 拿铁咖啡（NatieCoffee） | 10 | 橙汁（Orange） | 3 |   试用装饰者模式为该咖啡店设计一个程序以实现计算费用的功能，输出每种饮料的详细信息描述及花费。要求绘制类图并使用Java语言模拟实现。  结构：    代码：  BlueCoffee.java:  package test2\_4\_1;  public class BlueCoffee extends Component{  public String getDescription() {  return "蓝山咖啡";  }  public double getCost() {  return 100;  }  }  Client.java:  package test2\_4\_1;  public class Client {  public static void main(String[] args) {  Component c1,c2,c3;  c1=new BlueCoffee();  c2=new Milk(c1);  c3=new Orange(c2);  System.out.println("饮料："+c3.getDescription());  System.out.println("价格："+c3.getCost());  }  }  Component.java:  package test2\_4\_1;  public abstract class Component {  public abstract String getDescription();  public abstract double getCost();  }  Decorator.java:  package test2\_4\_1;  public abstract class Decorator extends Component{  private Component component;  public Decorator(Component co) {  this.component=co;  }    public double getCost() {  return component.getCost();  }    public String getDescription() {  return component.getDescription();  }  }  Milk.java:  package test2\_4\_1;  public class Milk extends Decorator{  public Milk(Component co) {  super(co);  }  public String getDescription() {  return super.getDescription()+addMilk();  }  public double getCost() {  return super.getCost()+addCost();  }    private double addCost() {  return 10;  }    private String addMilk() {  return "加牛奶";  }  }  NatieCoffee.java:  package test2\_4\_1;  public class NatieCoffee extends Component{  @Override  public String getDescription() {  return "拿铁咖啡";  }  @Override  public double getCost() {  return 14;  }    }  Orange.java:  package test2\_4\_1;  public class Orange extends Decorator{  public Orange(Component co) {  super(co);  }  private String addOrange() {  return "加橙汁";  }    private double addCost() {  return 3.5;  }    public String getDescription() {  return super.getDescription()+addOrange();  }    public double getCost() {  return super.getCost()+addCost();  }  }  类图：  测试：    （4.2）装饰模式实例之界面显示构件库 某软件公司基于面向对象技术开发了一套图形界面显示构件库VisualComponent。在使用该库构建某图形界面时，用户要求为界面定制一些特效显示效果，如带滚动条的窗体或透明窗体等。现使用装饰模式设计该构件库，绘制类图并编程模拟实现。  五、在某商品购物系统中，注册用户可对商品进行订购，商品的订购信息以订单的方式体现，假设只有用户才能对自己的订单进行修改，其他人都不能修改。试使用代理模式来设计并实现。  要求订单中需要包括订购的商品名称（productName），订购商品数量(productNum)，下单客户姓名（customerName）的信息。  要求先设计类图并使用面向对象语言进行实现。  结构：    代码：  Client.java:  package test2\_5\_1;  public class Client {  public static void main(String[] args) {  Order order = new Order("鞋子",120,"李四");  OrderMM orderMM = new OrderProxy(order);  orderMM.setProductNum(999, "张三");  System.out.println(order.toString());  orderMM.setProductNum(999, "李四");  System.out.println(order.toString());  }  }  Order.java:  package test2\_5\_1;  public class Order implements OrderMM {  private String productName;  private int productNum;  private String customerName;  public Order() {  }  public Order(String productName, int productNum, String customerName) {  this.productName = productName;  this.productNum = productNum;  this.customerName = customerName;  }  @Override  public String getProductName() {  return productName;  }  @Override  public void setProductName(String productName, String user) {  this.productName = productName;  System.out.println(customerName+"修改订单名字为："+productName+"成功");  }  @Override  public int getProductNum() {  return productNum;  }  @Override  public void setProductNum(int productNum, String user) {  this.productNum = productNum;  System.out.println(customerName+"修改订单数量为："+ productNum+"成功");  }  @Override  public String getCustomerName() {  // TODO 自动生成的方法存根  return customerName;  }  @Override  public void setCustomerName(String customerName, String user) {  // TODO 自动生成的方法存根    }    @Override  public String toString() {  return "修改后的订单信息：" +  "商品名称= '" + productName + '\'' +  ",商品数量= " + productNum;  }  }  OrderMM.java:  package test2\_5\_1;  public interface OrderMM {  String getProductName();  int getProductNum();  String getCustomerName();  void setProductName(String productName, String user);  void setProductNum(int productNum, String user);  void setCustomerName(String customerName, String user);  }  OrderProxy.java:  package test2\_5\_1;  public class OrderProxy implements OrderMM {  private Order order;  public OrderProxy(Order order) {  this.order = order;  }  @Override  public String getProductName() {  return this.order.getProductName();  }  @Override  public void setProductName(String productName, String user) {  if (user.equals(this.getCustomerName())) {  this.order.setProductName(productName, user);  } else {  System.out.println("对不起"+user+"无权修改商品名称");  }  }  @Override  public int getProductNum() {  return this.order.getProductNum();  }  @Override  public void setProductNum(int orderNum, String user) {  if (user.equals(this.getCustomerName())) {  this.order.setProductNum(orderNum, user);  } else {  System.out.println("对不起"+user+"无权修改商品数量");  }  }  @Override  public String getCustomerName() {  // TODO 自动生成的方法存根  return this.order.getCustomerName();  }  @Override  public void setCustomerName(String customerName, String user) {  // TODO 自动生成的方法存根  if (user.equals(this.getCustomerName())) {  this.order.setCustomerName(customerName, user);  } else {  System.out.println("对不起"+user+"无权修改");  }  }  }  类图：    测试： | | | | |
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
| 小结 | 学习掌握了适配器模式、桥接模式，外观模式，装饰模式和代理模式的的定义，结构及在特定环境下的应用。 | | | | |
|
|
|