

|  |
| --- |
| 实验报告  成绩（满分100） |
|  |

**《软件设计》实验报告**

**2019-2020学年第一学期**

姓名：刘文博

学号：201716040224

专业：软件工程

班级：1702班

2019年9月

**实验项目1．UML建模实验（6学时）**

**1．实验内容**

使用Rose、[StarUML](https://www.baidu.com/link?url=YKR9EiE37XXPeVP7raIo1rBTd0WzBlWSI2yl4cQSl-bGpx5xZOlipUFk22TY8R5rMjXP3VxsugGDWTjkdNckB6EowhsDjb7gUu70q-Ehy0_&wd=&eqid=b25725c30000067b0000000258fd6529" \t "_blank)、JUDE等绘制类图、顺序图，状态图，学习使用类图来构造软件的静态模型。

**2．基本要求**

（1）熟悉类图中各种模型元素的使用。

（2）熟悉在Rose、[StarUML](https://www.baidu.com/link?url=YKR9EiE37XXPeVP7raIo1rBTd0WzBlWSI2yl4cQSl-bGpx5xZOlipUFk22TY8R5rMjXP3VxsugGDWTjkdNckB6EowhsDjb7gUu70q-Ehy0_&wd=&eqid=b25725c30000067b0000000258fd6529" \t "_blank)、JUDE等开发环境中建立类图、顺序图，状态图。

**3．支撑的课程目标**

本实验项目可以支撑“课程目标1：能够对计算机软硬件开发项目进行UML建模，并对其进行评价和完善”。

本实验使学生建立软件系统的类图，顺序图，状态图，问题启发式引导学生分析系统，寻找问题的解决方案，加深相关知识点的理解，达到课程目标的要求。

1. **实验题目**

**（1）.** 用户通过登录界面(LoginForm)输入账号和密码，系统将输入的账号和密码与存储在数据库(User)表中的用户信息进行比较，验证用户输入是否正确，如果输入正确则进入主界面(MainForm)，否则提示“输入错误”。

**（2）.** 售票机相关部件的作用如下所述：

(1) 目的地键盘用来输入行程目的地的代码（例如，200表示总站）。

(2) 乘客可以通过车票键盘选择车票种类（单程票、多次往返票和座席种类）。

(3) 继续/取消键盘上的取消按钮用于取消购票过程，继续按钮允许乘客连续购买多张票。

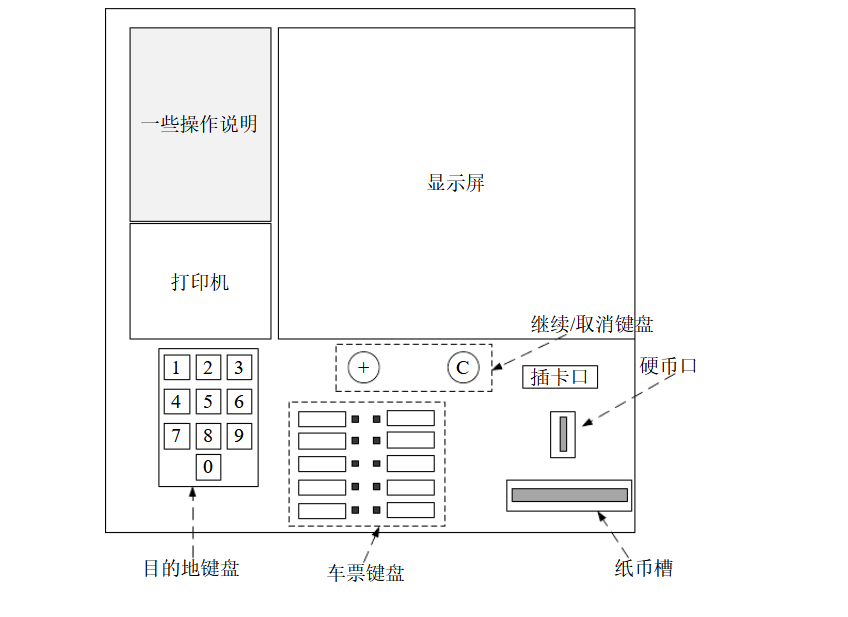
(4) 显示屏显示所有的系统输出和用户提示信息。

(5) 插卡口接受MCard（现金卡），硬币口和纸币槽接受现金。

(6) 打印机用于输出车票。

(7) 所有部件均可实现自检并恢复到初始状态。

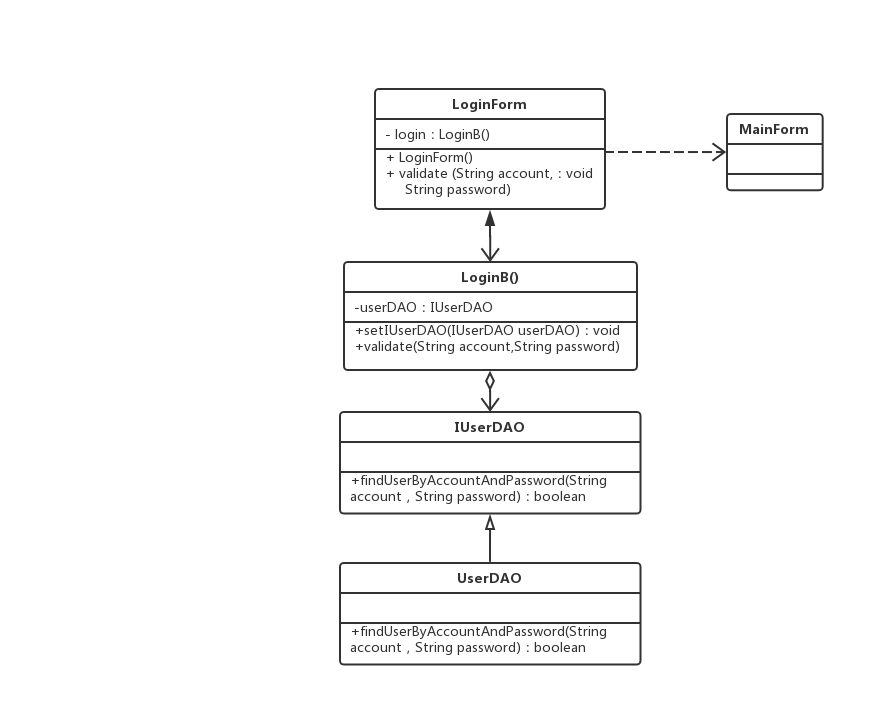
现采用面向对象方法开发该系统，使用UML进行建模，绘制该系统的核心类图并尽量分析出每一个类所包含的方法。



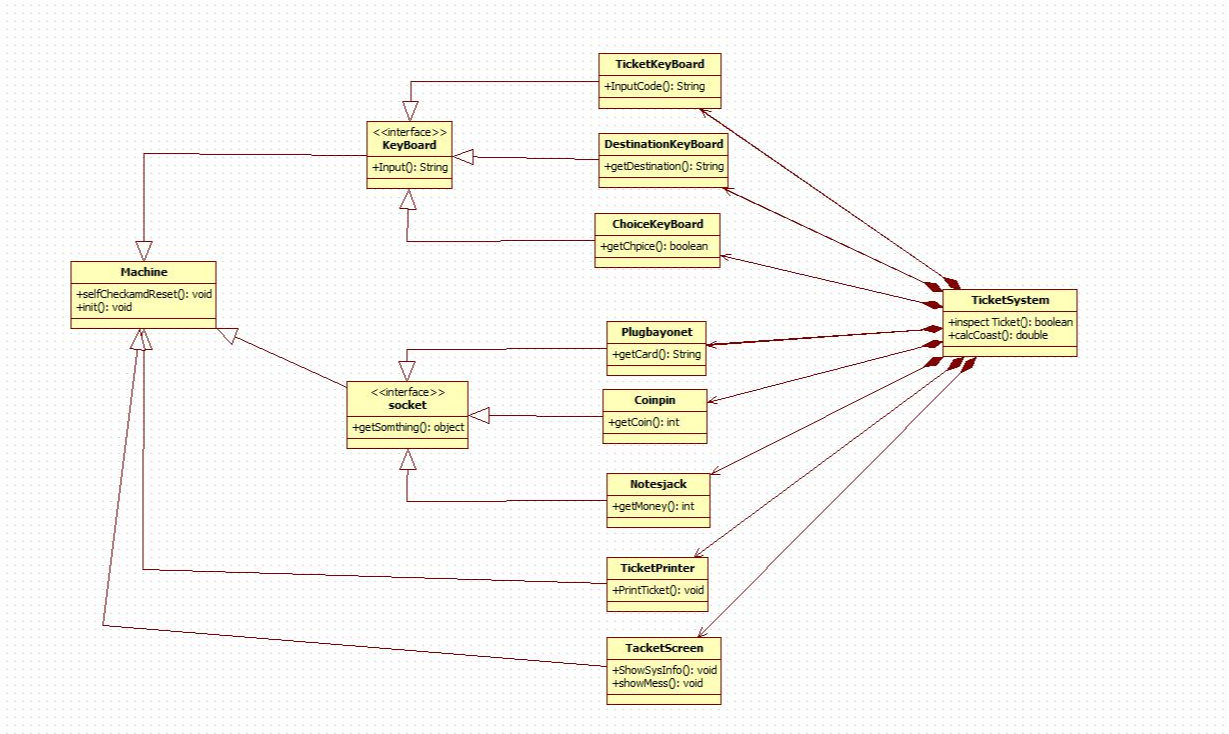
**（3）.** 某商场会员管理系统包含一个会员类(Member)，会员的基本信息包括会员编号、会员姓名、联系电话、电子邮箱、地址等，会员可分为金卡会员(GoldMember)和银卡会员(SilverMember)两种，不同类型的会员在购物时可以享受不同的折扣；每个会员可以拥有一个或多个订单(Order)，每一个订单又可以包含至少一条商品销售信息(ProductItem)，商品销售信息包括订单编号、商品编号、商品数量、商品单价和折扣等；每一条商品销售信息对应一类商品(Product)，商品信息包括商品编号、商品名称、商品单价、商品库存量、商品产地等。

1. **实验方法、实验过程及实验分析**

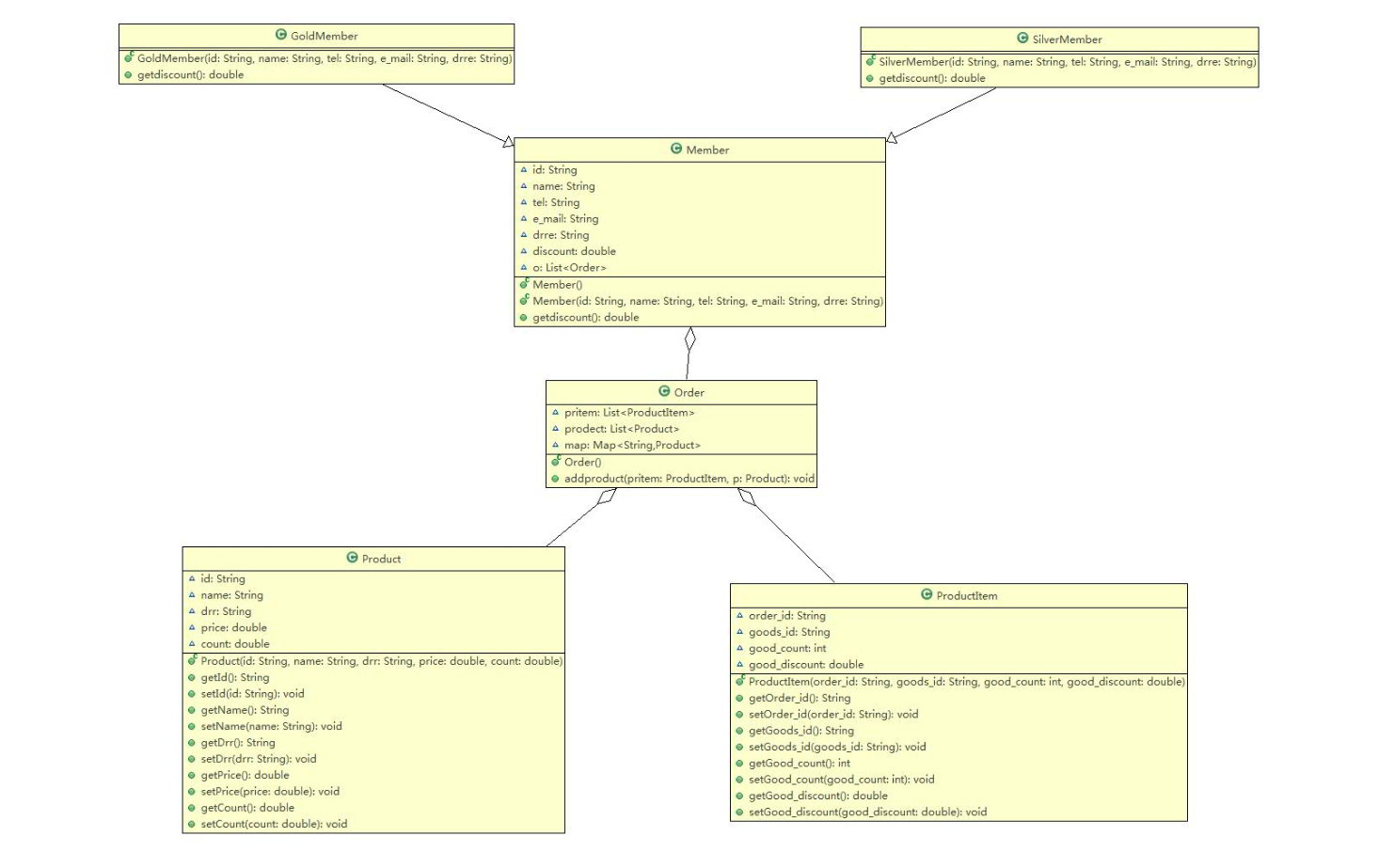
题目1



题目2



实验3



**6．实验总结**

1、学会了如何更加熟练，精确的绘制类图。

2、能将简单的需求用类图表示出来，提供抽象的数据访问接口（数据库），例如第三题。

4、熟练画类图时类与类之间的关系，如关联、聚合、组合、依赖、继承和实现的作用和意义。

**实验项目2．创建型模式实验（6学时）**

**1．实验内容**

使用创建型模式实现软件系统。

**2．基本要求**

（1）理解6种创建型模式。

（2）在软件项目开发过程中，使用创建型模式来设计软件。

**3．支撑的课程目标**

本实验项目可以支撑“课程目标2：能够在软件项目开发过程中,培养学生的创新精神”和“课程目标3：能够运用设计模式设计软件系统，支持复杂工程问题的求解”。

本实验使学生能分析软件系统，在软件项目的设计过程中使用创建型模式来设计软件，达到课程目标的要求。

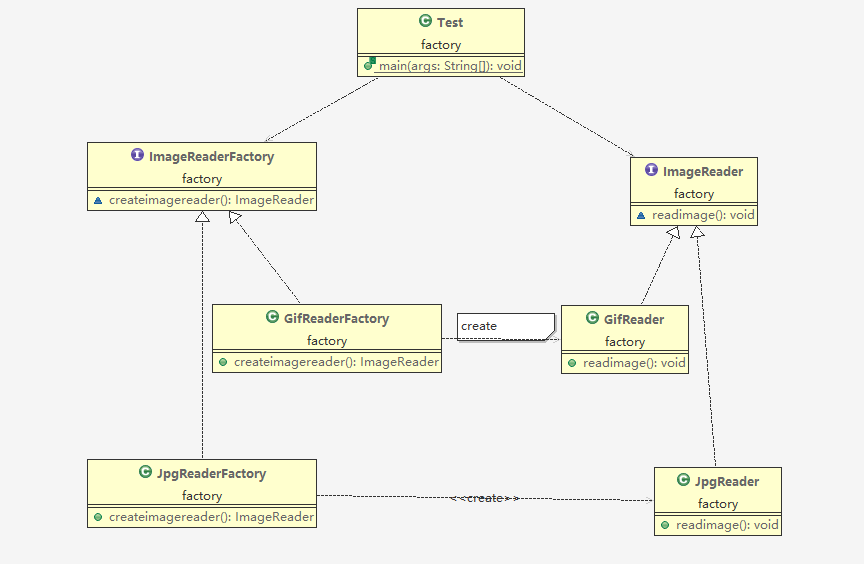
**4.实验题目**

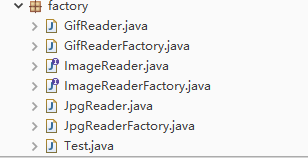
**（1）**现需要设计一个程序来读取多种不同类型的图片格式，针对每一种图片格式都设计一个图片读取器(ImageReader)，如GIF图片读取器(GifReader)用于读取GIF格式的图片、JPG图片读取器(JpgReader)用于读取JPG格式的图片。图片读取器对象通过图片读取器工厂ImageReaderFactory来创建，ImageReaderFactory是一个抽象类，用于定义创建图片读取器的工厂方法，其子类GifReaderFactory和JpgReaderFactory用于创建具体的图片读取器对象。试使用工厂方法模式设计该程序。

**（2）**使用单例模式的思想实现多例模式，确保系统中某个类的对象只能存在有限个，如两个或者三个，并编写代码实现一个多例类

**5.实验方法、实验过程及实验分析**

题目1：

做出类图如下



**抽象实体类:**

|  |
| --- |
| public interface ImageReader {  void readimage();  } |

**实体类：**

|  |
| --- |
| public class GifReader implements ImageReader{  @Override  public void readimage() {  System.*out*.println("读取GIF图片！");  }  }  public class JpgReader implements ImageReader{  @Override  public void readimage() {  System.out.println("读取JPG图片！");  }  } |

**抽象工厂类:**

|  |
| --- |
| public interface ImageReaderFactory {  ImageReader createimagereader();  } |

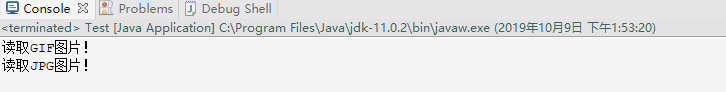
**工厂类:**

|  |
| --- |
| public class JpgReaderFactory implements ImageReaderFactory{  @Override  public ImageReader createimagereader() {  return new JpgReader();  }  }  public class GifReaderFactory implements ImageReaderFactory{  @Override  public ImageReader createimagereader() {  return new GifReader();  }  } |

**测试类:**

|  |
| --- |
| public class Test {  public static void main(String[] args) {  ImageReader reader1 = new GifReaderFactory().createimagereader();  ImageReader reader2 = new JpgReaderFactory().createimagereader();  reader1.readimage();  reader2.readimage();    }  } |

**运行结果：**



题目2：

**源代码：**

|  |
| --- |
| import java.util.ArrayList;  public class Money {  /\*\*  \* 多例类，确保系统中，某个对象的类只能存在有限个。  \*/  private static final int num = 3; //总定义实例存在的个数  private static int count = 0;  private static ArrayList<Money> m = null;  static {  m = new ArrayList<Money>(num);  }  private Money() {  }  /\*\* 普通模式，效率高\*/  public static Money getInstance() {  if (m.size() < num) {  while (m.size() < num) {  Money money = new Money();  m.add(money);  }  }  return m.get((count++) % 3);  } |

/\*\* 线程安全，效率低\*/

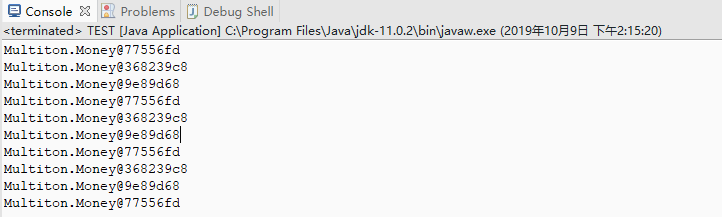
|  |
| --- |
| public static Money getInstance\_safe() {  if (m.size() < num) {  synchronized (m.getClass()) {  Money money;  if (m.size() < num) {  synchronized (m.getClass()) {  while (m.size() < num) {  money = new Money();  m.add(money);  }  }  }  }  }  return m.get((count++) % 3);  }  } |

**测试代码：**

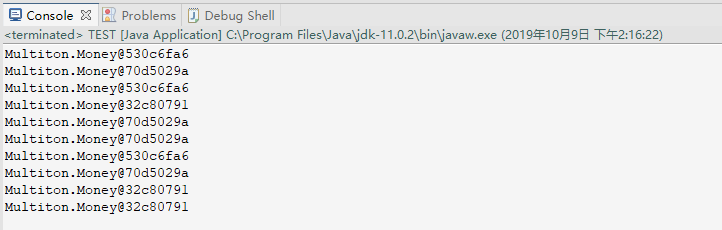
|  |
| --- |
| public class Test {  public static void main(String[] args) {  for(int i = 0 ; i < 10 ; i++) {  Money m = Money.getInstance();  System.out.println(m);  }      for (int i = 0; i < 20; i++) {  new Thread(new Runnable() {    @Override  public void run() {  Money m = Money.getInstance\_safe();  System.out.println(m);    }  }).start();  }    }  } |

**测试结果：**

普通模式结果



多线程结果



**6．实验总结**

工厂模式优缺点

经过本次工厂模式和单例模式的实验，通过自己动手编代码，是自己理解工厂单例模式机制，并且知

道工厂模式有适用性在下列情况下使用工厂方法模式：1.一个类不能预期它必须创建的对象的类2.一个类希望由其子类来指定它创建的对象优点1.通过消除应用特定的类的实例化，代码变得更灵活、更可复用2.代码只处理Product类的接口，能与任何支持该接口的ConcreteProduct类一起工作缺陷1.客户端可能只因实例化一个特定的ConcreteProduct而生成Creator 类的子类。

单例模式的优点

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。

单例模式的缺点

1、开销

虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要一些开销。可以通过使用静态初始化解决此问题。

2、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。

**实验项目3．结构型模式实验（6学时）**

**1．实验内容**

使用结构型模式设计软件系统。

**2．基本要求**

（1）熟悉7种结构型模式。

（2）使用结构型模式设计软件

**3．支撑的课程目标**

本实验项目可以支撑“课程目标2：能够在软件项目开发过程中,培养学生的创新精神”和“课程目标3：能够运用设计模式设计软件系统，支持复杂工程问题的求解”。

本实验使学生能分析软件系统，在软件项目的设计过程中使用结构型模式来设计软件，达到课程目标的要求。

**4.实验题目**

**1.**实现一个双向适配器实力,似的猫(Cat)可以学狗(Dog)叫,狗可以学猫抓老鼠,绘制相应类图并使用代码编程模拟.

**2.** 一个电器工厂可以产生多种类型的电器，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，模拟该场景，设计类图并编程实现

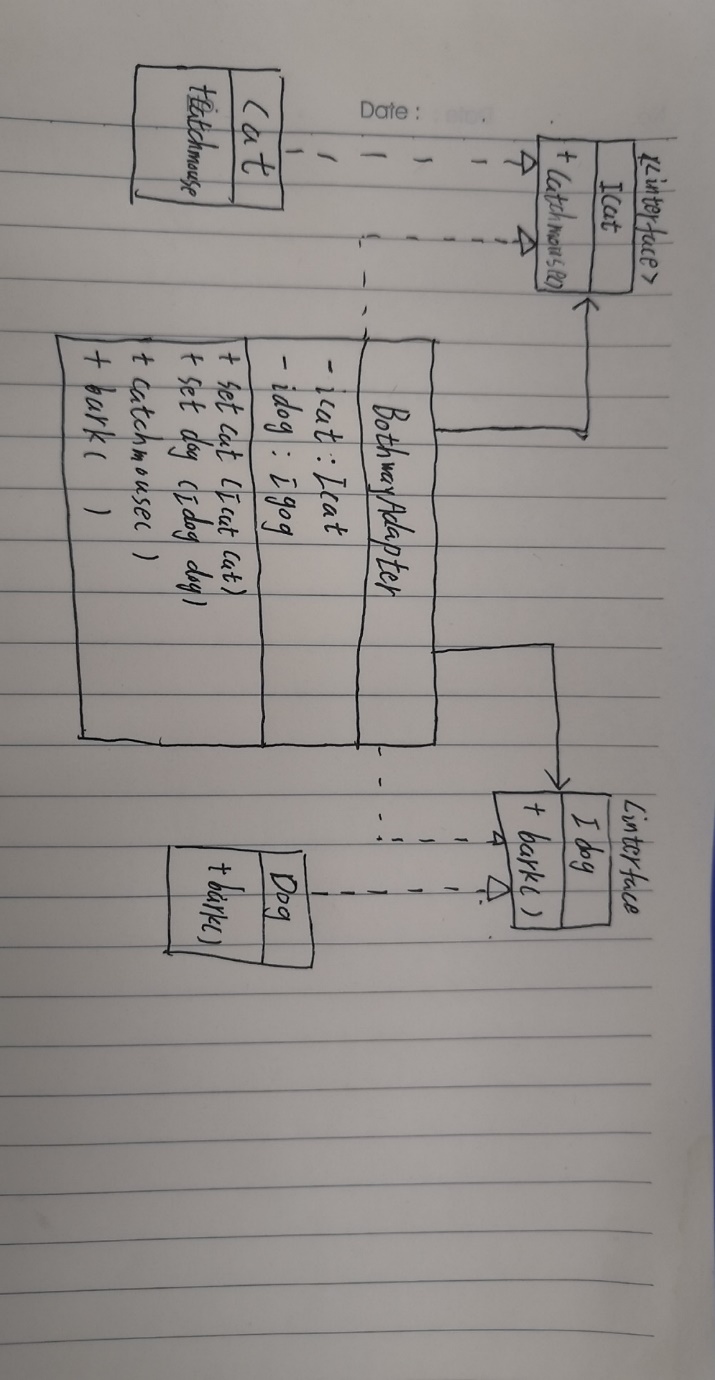
**3.** 在计算机主机（MainFrame）中，只需要按下主机的开机按钮(on())，就可以调用其他硬件设备和软件的启动方法，如内存(Memory)的自检(check())，CPU的运行(run())，硬盘(Harddisk)的读取(read())，操作系统(OS)的载入(load())等，如果某一过程发生错误，则计算机启动失败，使用外观模式模拟该过程绘制类图并编程实现。

（1）实现给出外观模式结构视图。

（2）给出该实例类图及代码实现。

**5.实验方法、实验过程及实验分析**

**1.类图**



**抽象类:**

|  |
| --- |
| public interface ICat {  void catlook();  void catchmouse();  }  public interface IDog {  void look();  void bark();  } |

**实体类:**

|  |
| --- |
| public class Cat implements ICat {  @Override  public void catlook() {  System.out.println("我是一只猫!");  }  @Override  public void catchmouse() {  System.out.println("我会抓老鼠!");  }  }  public class Dog implements IDog {  @Override  public void look() {  System.out.println("我是一只狗!");  }  @Override  public void bark() {  System.out.println("汪汪汪!!!!");  }  } |

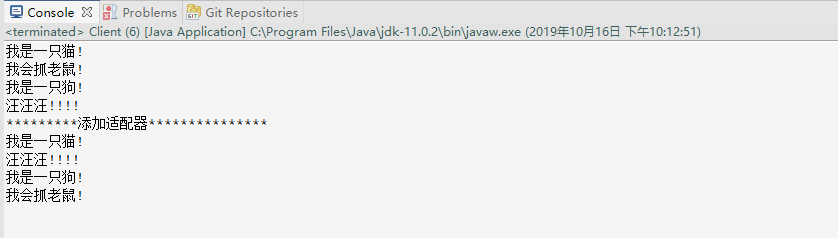
**适配器:**

|  |
| --- |
| public class TwowayAdapter implements ICat, IDog {  ICat cat ;  IDog dog ;    public TwowayAdapter(ICat cat) {  this.cat = cat;  }  public TwowayAdapter(IDog dog) {  this.dog = dog;  }    @Override  public void bark() {  cat.catchmouse();    }  @Override  public void catchmouse() {  dog.bark();    }  @Override  public void look() {  System.out.println("我是一只狗!");  }  @Override  public void catlook() {  System.out.println("我是一只猫!");    }  } |

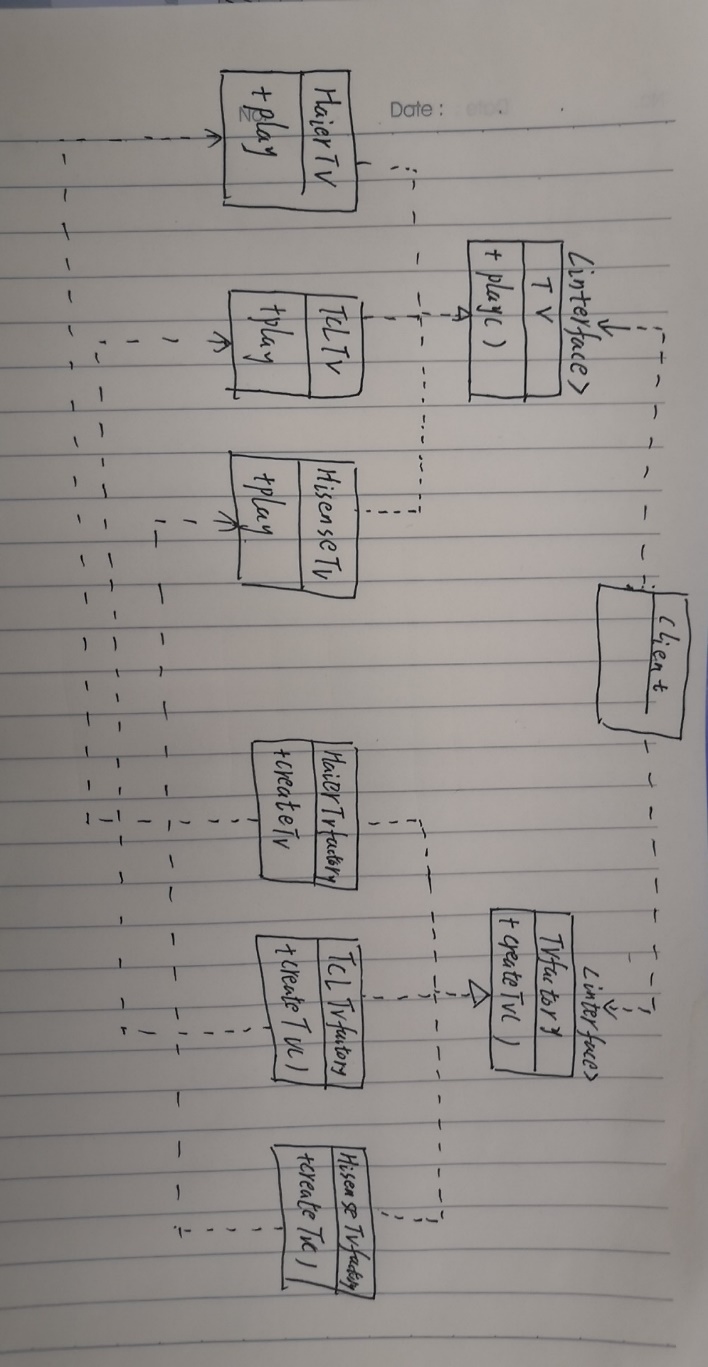
**测试类:**

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Cat cat = new Cat();  Dog dog = new Dog();  cat.catlook();  cat.catchmouse();  dog.look();  dog.bark();  System.out.println("\*\*\*\*\*\*\*\*\*添加适配器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  ICat cat1 = new TwowayAdapter(dog);  cat1.catlook();  cat1.catchmouse();  IDog dog1 = new TwowayAdapter(cat);  dog1.look();  dog1.bark();  }  } |

**运行结果:**



**2.类图**



**抽象产品类:**

|  |
| --- |
| public interface Television {  public void play();  }  public interface AirConditioner {  public void changeTemperature();  } |

**抽象工厂类:**

|  |
| --- |
| public interface EFactory {  public Television produceTelevision();  public AirConditioner produceAirConditioner();  } |

**具体产品类:**

|  |
| --- |
| public class HaierTelevision implements Television {  public void play() {  System.out.println("海尔电视机播放中.");  }  }  public class TCLTelevision implements Television {  public void play() {  System.out.println("TCL电视机播放中.");  }  }  public class HaierAirConditioner implements AirConditioner {  public void changeTemperature() {  System.out.println("海尔空调控温中");  }  }  public class HaierAirConditioner implements AirConditioner {  public void changeTemperature() {  System.out.println("TCL空调控温中");  }  } |

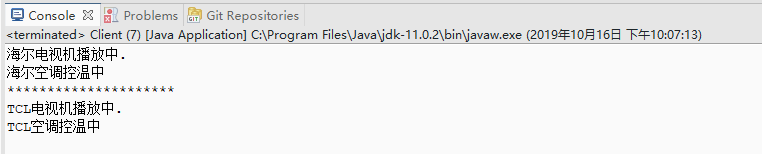
**具体工厂类:**

|  |
| --- |
| public class HaierFactory implements EFactory {  public Television produceTelevison() {  return new HaierTelevision();  }  public AirConditioner produceAirConditioner {  return new HaierAirConditioner();  }  }  public class TCLFactory implements EFactory {  public Television produceTelevison() {  return new TCLTelevision();  }  public AirConditioner produceAirConditioner {  return new TCLAirConditioner();  }  } |

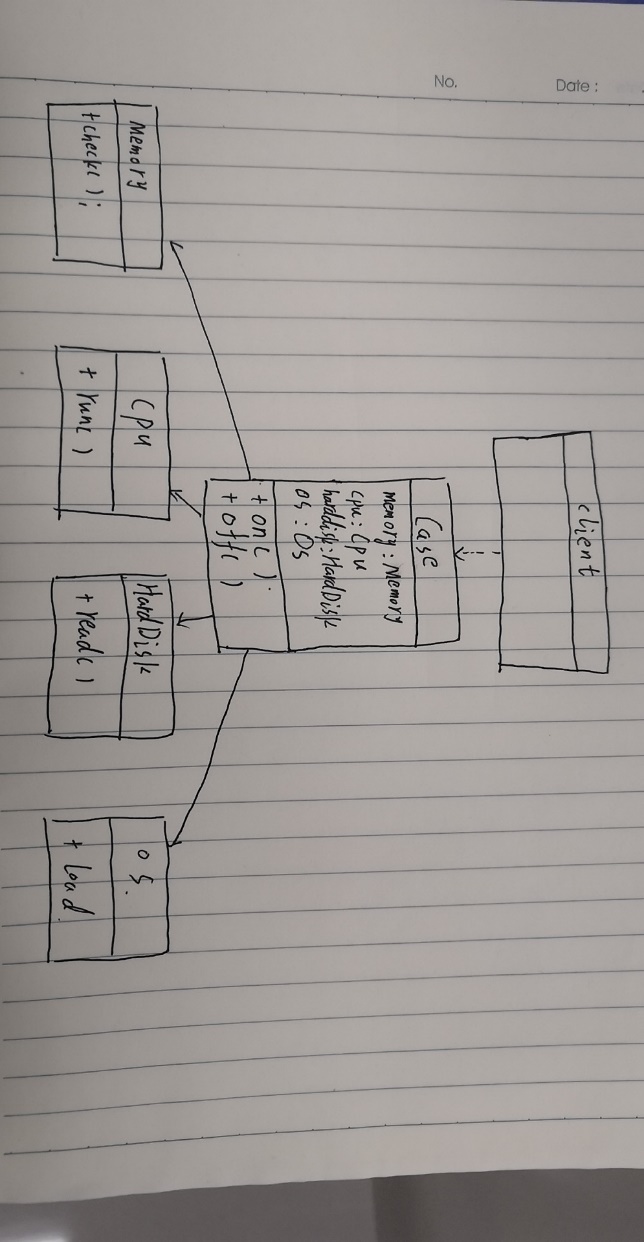
**测试代码:**

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Television tv;  AirConditioner airConditioner;  EFactory factory = new HaierFactory();  tv = factory.produceTelevision();  airConditioner = factory.produceAirConditioner();  tv.play();  airConditioner.changeTemperature();    System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");    factory = new TCLFactory();  tv = factory.produceTelevision();  airConditioner = factory.produceAirConditioner();  tv.play();  airConditioner.changeTemperature();  }  } |

**实验结果:**



**3.类图**



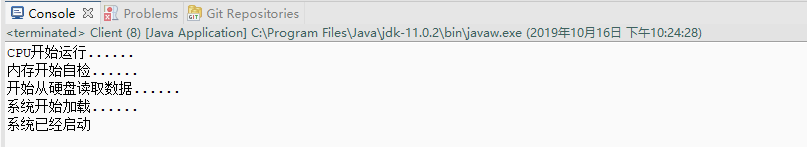
**实体类:**

|  |
| --- |
| public class CPU {  void run() {  System.out.println("CPU开始运行......");  }  }  public class HardDisk {  void read() {  System.out.println("开始从硬盘读取数据......");  }  }  public class Memory {  void check() {  System.out.println("内存开始自检......");  }  }  public class OS {  void load() {  System.out.println("系统开始加载......");  }  }  public class Mainframe {  CPU cpu = new CPU();  Memory memory = new Memory();  OS os = new OS();  HardDisk harddisk = new HardDisk();    public void on() {  cpu.run();  memory.check();  harddisk.read();  os.load();  System.out.println("系统已经启动");  }    public void off() {  System.out.println("系统已经关闭");  }  } |

**测试类:**

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Mainframe mainfram = new Mainframe();  mainfram.on();  }  } |

**实验结果:**



**6．实验总结**

对象适配器和类适配器其实算是同一种思想，只不过实现方式不同。 根据合成复用原则，组合大于继承， 所以它解决了类适配器必须继承src的局限性问题，也不再强求dst必须是接口。 同样的它使用成本更低，更灵活。

抽象工厂模式最大的好处是易于交换产品系列，由于具体工厂类， 在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需要改变具体工厂即可使用不同的产品配置。不管是任何人的设计都无法去完全防止需求的更改，或者项目的维护，那么我们的理想便是让改动变得最小、最容易，例如我现在要更改以上代码的数据库访问时，只需要更改具体的工厂即可。

抽象工厂模式的另一个好处就是它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操作实例，产品实现类的具体类名也被具体的工厂实现类分离，不会出现在客户端代码中。就像我们上面的例子，客户端只认识IUser和IDepartment，至于它是MySQl里的表还是Oracle里的表就不知道了。

通过引入外观模式，外观类，客户端代码只和外观类直接通信，代码变得简单。实现了子系统与客户之间的松耦合关系。客户端依然可以直接访问子系统。但是违反了开闭原则

**实验项目4．综合实践练习实验（6学时）**

**1．实验内容**

综合多种设计模式进行软件设计。

**2．基本要求**

（1）熟悉行为型模式，并使用行为型模式综合进行软件设计。

（2）综合多种设计模式进行软件设计，并分析其优缺点。

**3．支撑的课程目标**

本实验项目可以支撑“课程目标3：能够运用设计模式设计软件系统，支持复杂工程问题的求解”和“课程目标4：能够及时跟踪软件工程领域发展状况，对当前的热点问题及时跟踪并发表自己见解”。

本实验通过综合实验使学生能分析软件系统，在软件项目的设计过程中综合使用各种模式来设计软件，达到课程目标的要求。

**4.实验题目**

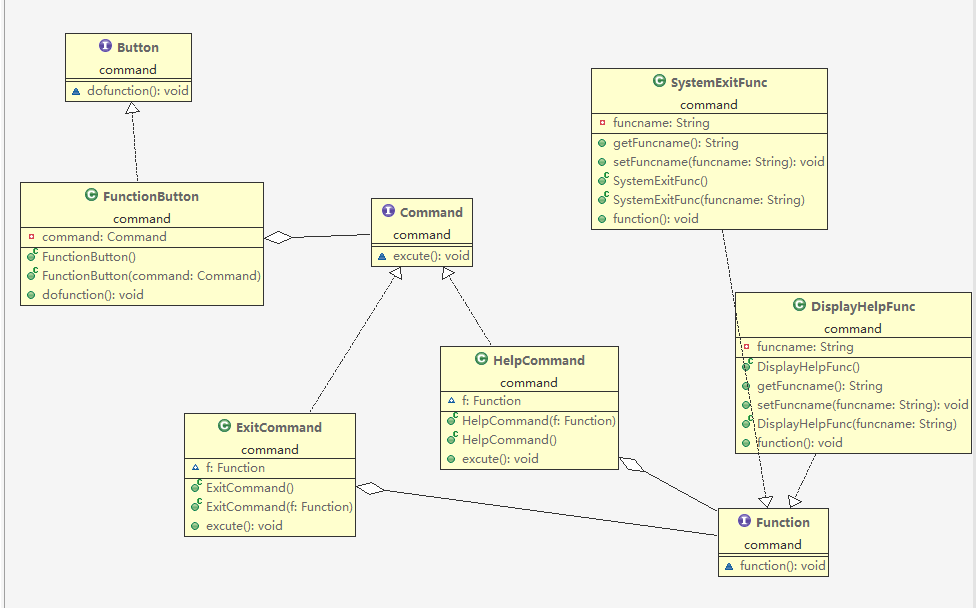
1**.**为了使用户使用方便,某系统提供了一系列功能键,用户可以自定义功能键的功能,如功能键FunctionButton可以用于退出系统(SystemExitFunction),也可以用于打开帮助界面(DispalyHelpClass).用户可以通过修改配置文件来改变功能键的用途,现使用命令模式来设计该系统,使功能键于功能类解耦,相同的功能键可以对应不同的功能.

2.某在线游戏支持多人联机对战,每个玩家都可以加入某一战队组成联盟,当战队中的某一成员收到敌人的攻击是将给所有的盟友发送通知,盟友收到通知后将做出响应.使用观察者模式设计并实现该过程.

3.某系统需要对重要数据进行加密,并提供了几种加密方案(如凯撒加密,DES加密等),对该加密模块济宁设计,使得用户可以动态选择加密方式,要求绘制类图并编程模拟实现.

**5.实验方法、实验过程及实验分析**

(1)类图如下:



**调用对象:**

|  |
| --- |
| public interface Button {    void dofunction();  }  public class FunctionButton implements Button {  private Command command;    public FunctionButton() {  }  public FunctionButton(Command command) {  super();  this.command = command;  }  @Override  public void dofunction() {  command.excute();    }  } |

**命令对象(抽象命令/具体命令):**

|  |
| --- |
| public interface Command {  void excute();  }  public class ExitCommand implements Command {  Function f ;  public ExitCommand() {  f = new SystemExitFunc();    }  public ExitCommand(Function f) {  super();  this.f = f;  }    @Override  public void excute() {  f.function();  }  }  public class HelpCommand implements Command {    Function f ;    public HelpCommand(Function f) {  super();  this.f = f;  }  public HelpCommand() {  f = new DisplayHelpFunc();  }  @Override  public void excute() {  f.function();    }  } |

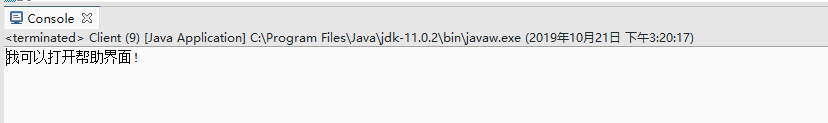
**具体执行着:**

|  |
| --- |
| public interface Function {  void function();  }  public class DisplayHelpFunc implements Function {  private String funcname ;    public DisplayHelpFunc() {  }    public String getFuncname() {  return funcname;  }  public void setFuncname(String funcname) {  this.funcname = funcname;  }  public DisplayHelpFunc(String funcname) {  super();  this.funcname = funcname;  }  @Override  public void function() {  System.out.println("我可以打开帮助界面!");  }  }  public class SystemExitFunc implements Function {  private String funcname ;    public String getFuncname() {  return funcname;  }  public void setFuncname(String funcname) {  this.funcname = funcname;  }  public SystemExitFunc() {  super();  }  public SystemExitFunc(String funcname) {  super();  this.funcname = funcname;  }    @Override  public void function() {  System.out.println("我用于退出系统!");    }  } |

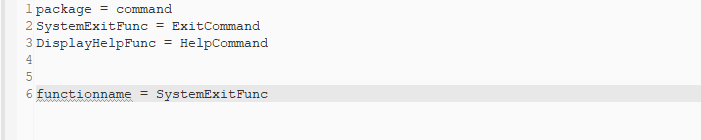
**运行结果:**

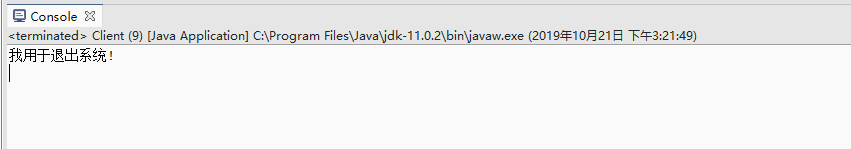
1.





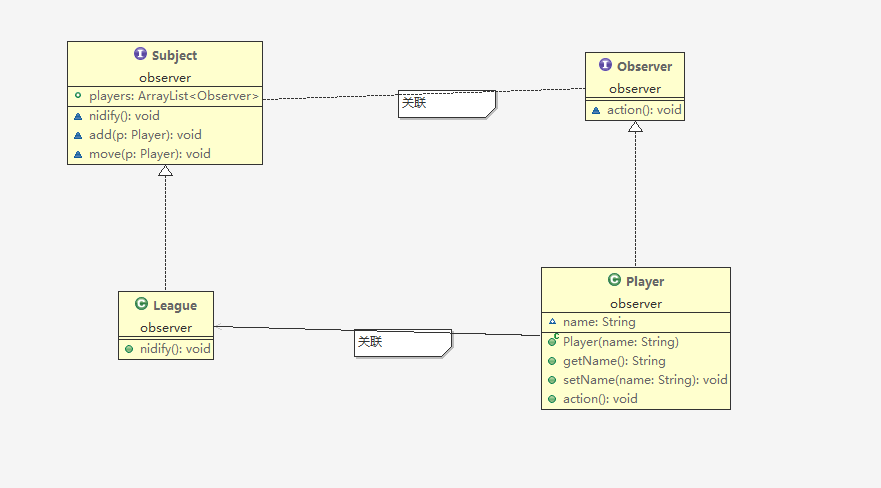
2.





2.

**类图如下:**



**代码:**

**目标类**

|  |
| --- |
| public interface Subject {  ArrayList<Observer> players = new ArrayList<Observer>();  void nidify();    default void add(Player p) {  players.add(p);  }  default void move(Player p) {  players.remove(p);  }  }  public class League implements Subject{  @Override  public void nidify() {  System.out.println("发现有人受伤!!");    for(Observer p : players) {  p.action();  }    }  } |

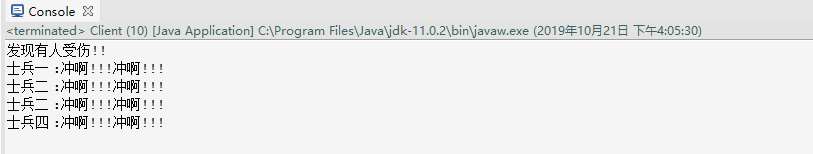
**观察者类:**

|  |
| --- |
| public interface Observer {  void action();  }  public class Player implements Observer{    String name ;    public Player(String name) {  this.name = name;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }    @Override  public void action() {  System.out.println(getName() + " :冲啊!!!冲啊!!!");    }  } |

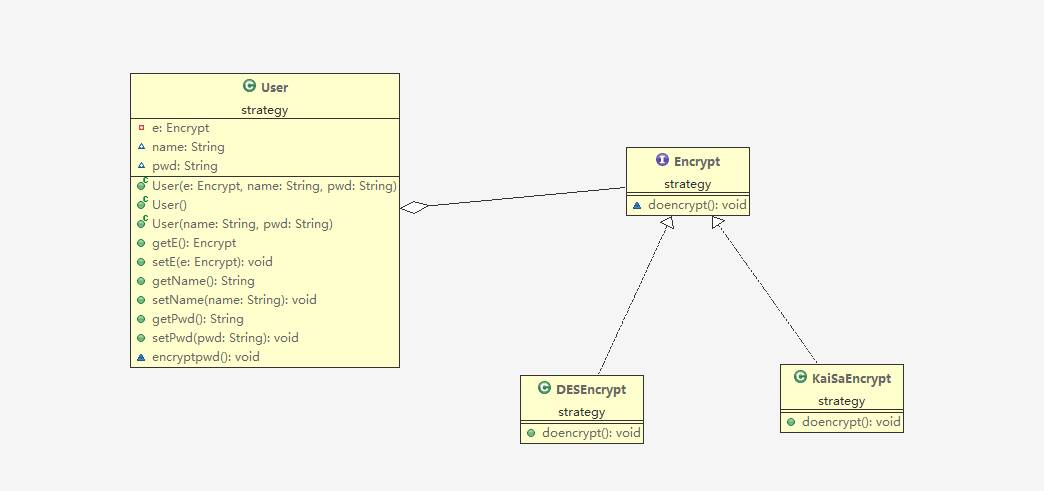
**测试代码:**

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Player p1 = new Player("士兵一");  Player p2 = new Player("士兵二");  Player p3 = new Player("士兵三");  Player p4 = new Player("士兵四");    Subject league = new League();  league.add(p1);  league.add(p2);  league.add(p2);  league.add(p4);    league.nidify();    }  } |

**运行结果:**



**3.类图如下:**



**代码如下:**

**加密方法:**

|  |
| --- |
| public interface Encrypt {  void doencrypt();  }  public class KaiSaEncrypt implements Encrypt {  @Override  public void doencrypt() {  System.out.println("真在进行凯撒加密!");  }  }  public class DESEncrypt implements Encrypt {  @Override  public void doencrypt() {  System.out.println("正在进行DES加密!");    }  } |

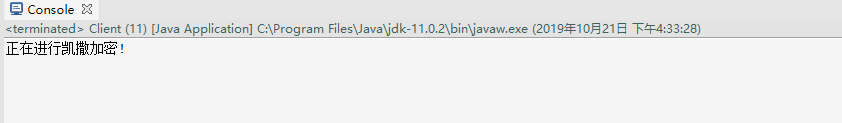
**用户类:**

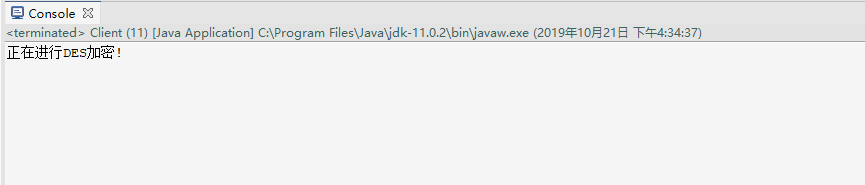
|  |
| --- |
| public class User {  private Encrypt e;  String name;  String pwd;  public User(Encrypt e, String name, String pwd) {  super();  this.e = e;  this.name = name;  this.pwd = pwd;  }  public User() {  }  public User(String name, String pwd) {  super();  this.name = name;  this.pwd = pwd;  }  public Encrypt getE() {  return e;  }  public void setE(Encrypt e) {  this.e = e;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getPwd() {  return pwd;  }  public void setPwd(String pwd) {  this.pwd = pwd;  }  void encryptpwd() {  e.doencrypt();  }    } |

**测试代码:**

|  |
| --- |
| public class Client {  /\*\*  \* 策略模式  \* @param args  \*/  public static void main(String[] args) {  Encrypt e = new KaiSaEncrypt();  User user = new User("张三", "123456");  user.setE(e);  user.encryptpwd();  }  } |

**实验截图:**



****

**6．实验总结**

命令（CommandPattern）模式的定义如下：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。但是也可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。

总结：调用者与接收者通过命令对象间接沟通，就类似于中间件，负载均衡等。

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。