

|  |
| --- |
| 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10月

**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**

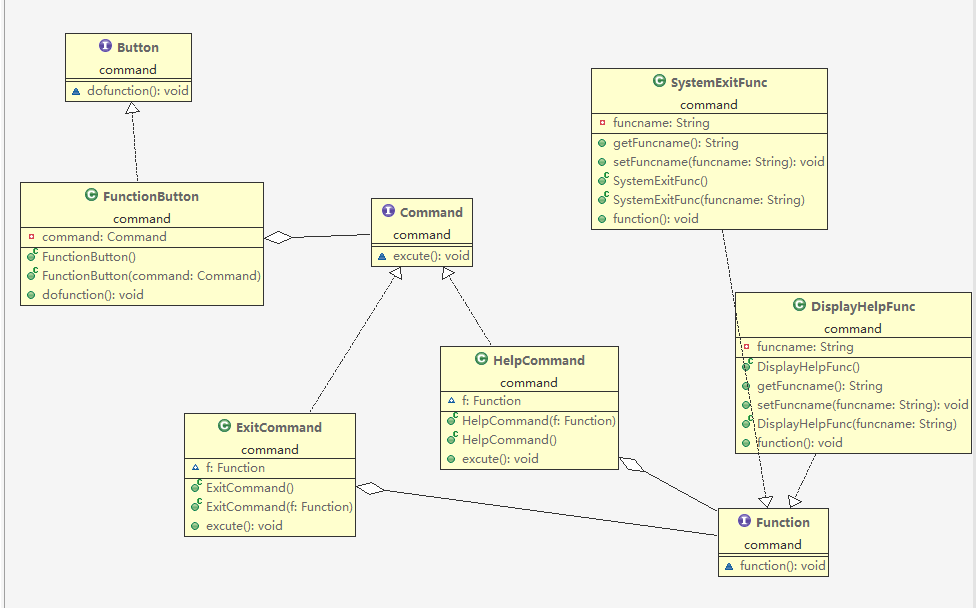
1**.**为了使用户使用方便,某系统提供了一系列功能键,用户可以自定义功能键的功能,如功能键FunctionButton可以用于退出系统(SystemExitFunction),也可以用于打开帮助界面(DispalyHelpClass).用户可以通过修改配置文件来改变功能键的用途,现使用命令模式来设计该系统,使功能键于功能类解耦,相同的功能键可以对应不同的功能.

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();    }  } |

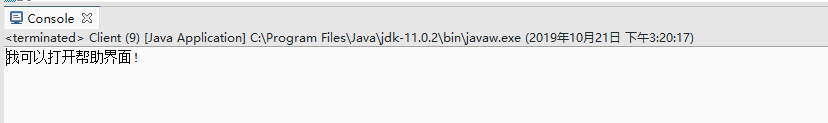
**具体执行着:**

|  |
| --- |
| 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!");    }  } |

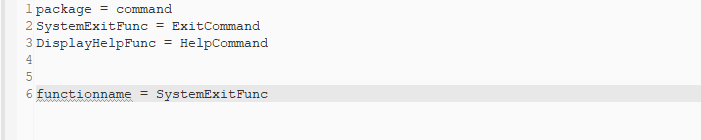
**运行结果:**

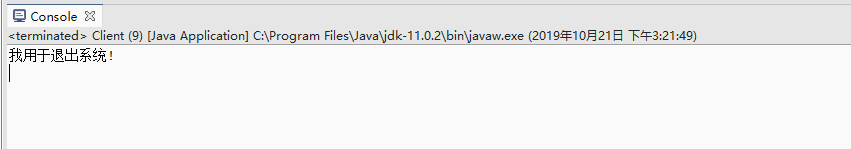
1.





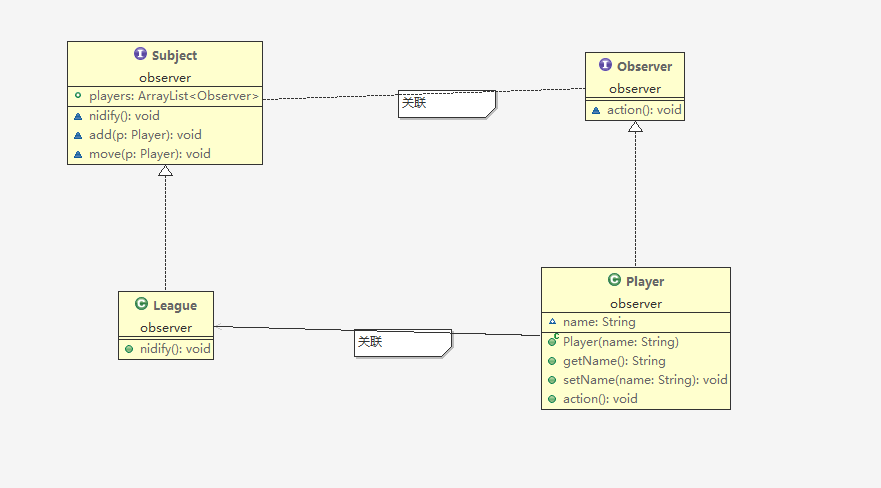
2.





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();  }    }  } |

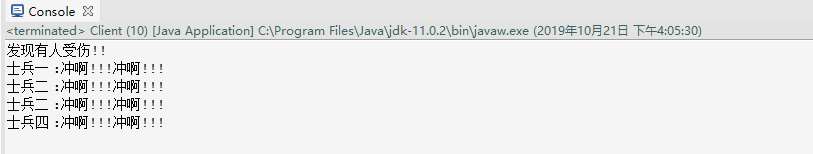
**观察者类:**

|  |
| --- |
| 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!!!");    }  } |

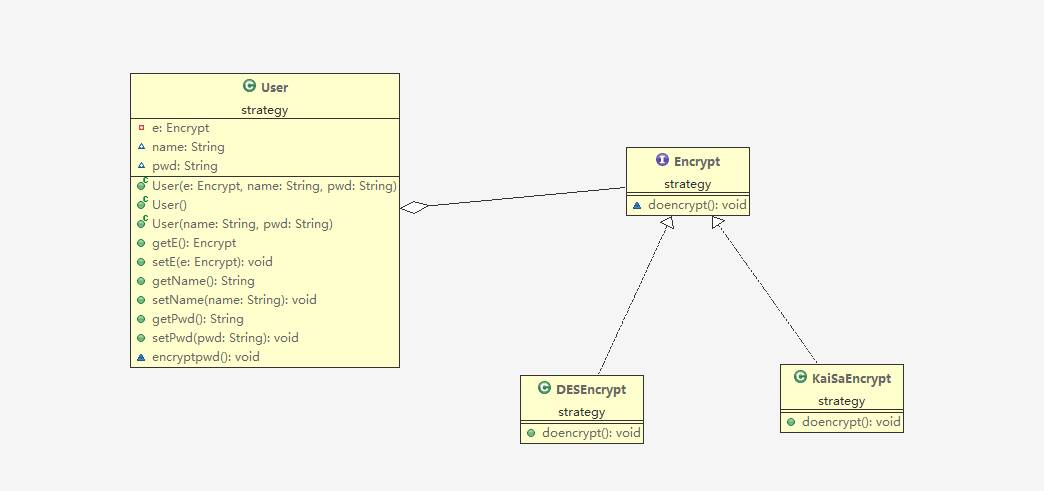
**测试代码:**

|  |
| --- |
| 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!");    }  } |

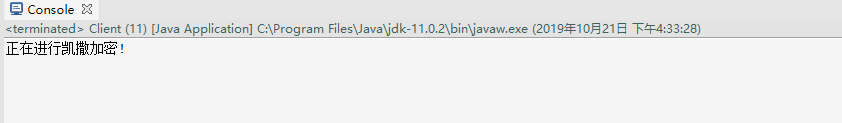
**用户类:**

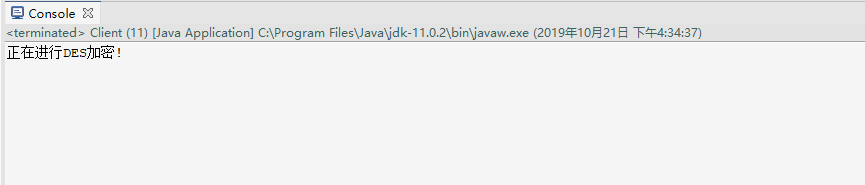
|  |
| --- |
| public class User {  private Encrypt e;  String name;  String pwd;  public User(Encrypt e, String name, String pwd) {  super();  this.e = e;  this.name = name;  this.pwd = pwd;  }  public User() {  }  public User(String name, String pwd) {  super();  this.name = name;  this.pwd = pwd;  }  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。