**《软件工程》**

**实验报告七 ：设计模式以及构件设计**

**姓 名： 郝舒森 学 号： 202210120305**

**院 系： 计算机与信息学院 专 业： 计算机科学与技术**

**实 验 室： J-1307 实验日期： 12.3**

**总评成绩： 审阅教师：**

### 一、实验目的

1. 了解软件工程工具生成代码的方法，理解代码自动生成的过程，破除自动生成代码“黑科技”

2. 掌握构件设计方法

3. 学习Bridge设计模式，体会设计模式的作用

### 二、实验环境

PlantUML，idea，java17

### 三、实验要求

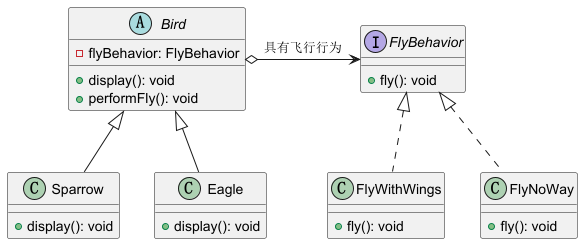
1.参考第7章 设计概念（design principle &pattern）（4学时），学习Bridge设计模式的使用；

2.学习 Rose 生成代码的方法，理解代码自动生成原理

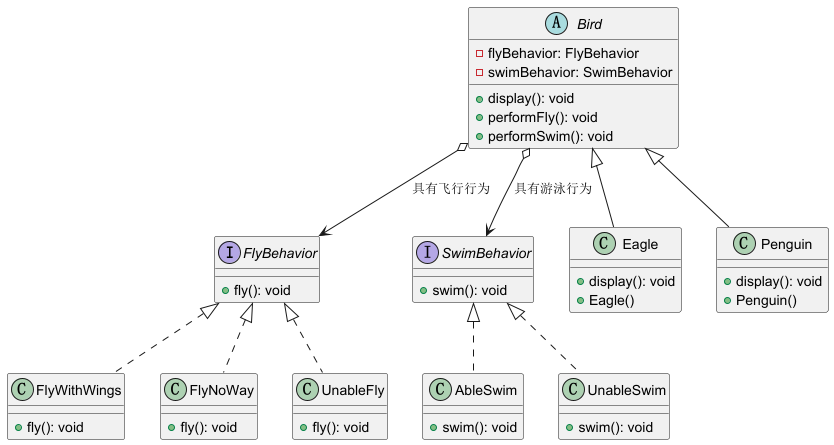
3.文件以“学号-姓名-软件工程实验七.rar”的方式命名，把模型文件和工程实现代码打包，提交到长江雨课堂软件工程实验七；

四、实验内容

问题一类图



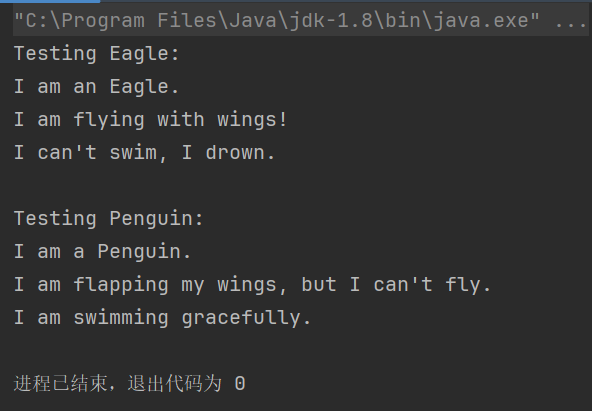
问题二拓展后类图：



问题二代码：

// 飞行行为接口及实现类  
interface FlyBehavior {  
 void fly();  
}  
  
class FlyWithWings implements FlyBehavior {  
 @Override  
 public void fly() {  
 System.*out*.println("I am flying with wings!");  
 }  
}  
  
class FlyNoWay implements FlyBehavior {  
 @Override  
 public void fly() {  
 System.*out*.println("I can't fly.");  
 }  
}  
  
class UnableFly implements FlyBehavior {  
 @Override  
 public void fly() {  
 System.*out*.println("I am flapping my wings, but I can't fly.");  
 }  
}  
  
// 游泳行为接口及实现类  
interface SwimBehavior {  
 void swim();  
}  
  
class AbleSwim implements SwimBehavior {  
 @Override  
 public void swim() {  
 System.*out*.println("I am swimming gracefully.");  
 }  
}  
  
class UnableSwim implements SwimBehavior {  
 @Override  
 public void swim() {  
 System.*out*.println("I can't swim, I drown.");  
 }  
}  
  
  
  
  
// 鸟类的抽象类  
abstract class Bird {  
 protected FlyBehavior flyBehavior;  
 protected SwimBehavior swimBehavior;  
  
 public Bird(FlyBehavior flyBehavior, SwimBehavior swimBehavior) {  
 this.flyBehavior = flyBehavior;  
 this.swimBehavior = swimBehavior;  
 }  
  
 public abstract void display();  
  
 public void performFly() {  
 flyBehavior.fly();  
 }  
  
 public void performSwim() {  
 swimBehavior.swim();  
 }  
}  
  
// 具体的鸟类：老鹰  
class Eagle extends Bird {  
 public Eagle() {  
 super(new FlyWithWings(), new UnableSwim()); // 老鹰会飞，但不会游泳  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("I am an Eagle.");  
 }  
}  
  
// 具体的鸟类：企鹅  
class Penguin extends Bird {  
 public Penguin() {  
 super(new UnableFly(), new AbleSwim()); // 企鹅不会飞，但能游泳  
 }  
  
 @Override  
 public void display() {  
 System.*out*.println("I am a Penguin.");  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Bird eagle = new Eagle();  
 Bird penguin = new Penguin();  
  
 // 测试老鹰  
 System.*out*.println("Testing Eagle:");  
 eagle.display();  
 eagle.performFly();  
 eagle.performSwim();  
  
 System.*out*.println();  
  
 // 测试企鹅  
 System.*out*.println("Testing Penguin:");  
 penguin.display();  
 penguin.performFly();  
 penguin.performSwim();  
 }  
}

问题二代码运行结果：



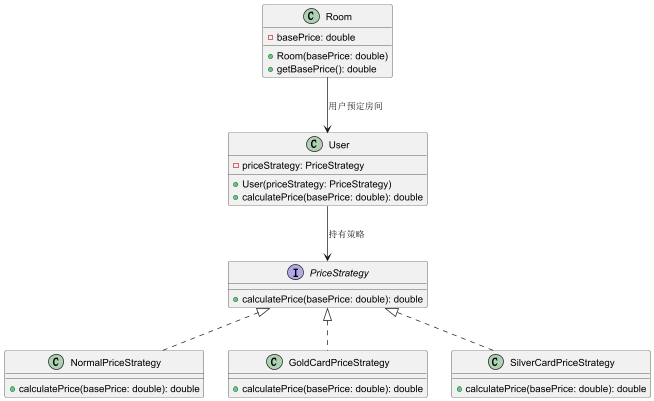
问题三设计思路：

使用 **策略模式（Strategy Pattern）** 来根据不同的用户类型计算房价。策略模式的核心思想是将不同的算法封装在独立的策略类中，客户端可以根据需要选择不同的策略来完成某项功能。

**开闭原则（Open/Closed Principle）**：通过策略模式，当我们需要添加新的用户类型时，只需要扩展新的策略类，而无需修改原有的类。

**单一职责原则（Single Responsibility Principle）**：每个类负责自己的独立职责。例如，房价计算类只负责计算房价，用户类只负责表示用户信息。

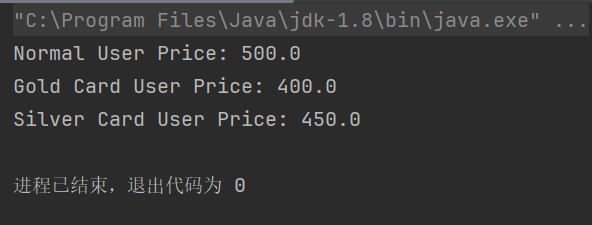
问题三类图：



问题三代码：

// 房价策略接口  
interface PriceStrategy {  
 double calculatePrice(double basePrice);  
}  
  
// 普通用户策略：全价  
class NormalPriceStrategy implements PriceStrategy {  
 @Override  
 public double calculatePrice(double basePrice) {  
 return basePrice; // 不打折，原价  
 }  
}  
  
// 金卡用户策略：8折  
class GoldCardPriceStrategy implements PriceStrategy {  
 @Override  
 public double calculatePrice(double basePrice) {  
 return basePrice \* 0.8; // 打8折  
 }  
}  
  
// 银卡用户策略：9折  
class SilverCardPriceStrategy implements PriceStrategy {  
 @Override  
 public double calculatePrice(double basePrice) {  
 return basePrice \* 0.9; // 打9折  
 }  
}  
  
  
  
// 用户类，持有一个策略对象  
class User {  
 private PriceStrategy priceStrategy;  
  
 public User(PriceStrategy priceStrategy) {  
 this.priceStrategy = priceStrategy;  
 }  
  
 // 计算价格  
 public double calculatePrice(double basePrice) {  
 return priceStrategy.calculatePrice(basePrice);  
 }  
}  
  
// 房间类，表示一个房间  
class Room {  
 private double basePrice;  
  
 public Room(double basePrice) {  
 this.basePrice = basePrice;  
 }  
  
 public double getBasePrice() {  
 return basePrice;  
 }  
}  
  
  
public class HotelReservationSystem {  
 public static void main(String[] args) {  
 // 创建不同的策略对象  
 PriceStrategy normalStrategy = new NormalPriceStrategy();  
 PriceStrategy goldCardStrategy = new GoldCardPriceStrategy();  
 PriceStrategy silverCardStrategy = new SilverCardPriceStrategy();  
  
 // 创建不同的用户对象  
 User normalUser = new User(normalStrategy);  
 User goldCardUser = new User(goldCardStrategy);  
 User silverCardUser = new User(silverCardStrategy);  
  
 // 创建一个房间，基本价格为 500  
 Room room = new Room(500);  
  
 // 打印不同用户的房价  
 System.*out*.println("Normal User Price: " + normalUser.calculatePrice(room.getBasePrice()));  
 System.*out*.println("Gold Card User Price: " + goldCardUser.calculatePrice(room.getBasePrice()));  
 System.*out*.println("Silver Card User Price: " + silverCardUser.calculatePrice(room.getBasePrice()));  
 }  
}

问题三运行结果：



### 五、实验结论

通过本次实验，我深入理解了桥接模式和策略模式的实际应用，并在实际编码中加深了对面向对象设计原则的理解。在酒店预订系统的设计中，我通过策略模式成功实现了不同用户类型计算房价的需求，进一步掌握了如何通过接口和实现类解耦不同的算法，以便于将来在不修改原有代码的基础上扩展新的功能。此外，通过这个实验，我体验了如何利用设计模式提升代码的灵活性和可扩展性，使得代码能够更好地适应未来可能发生的变化。策略模式的使用让我意识到，通过将具体算法与上下文环境分离，能够使代码更具通用性和可维护性。在桥接模式的练习中，我更好地理解了如何将变化的部分（例如飞行行为和游泳行为）与不变的部分（例如鸟类的基本特性）分离，使得在增加新的行为或特性时，不会影响到已有的类结构。这些设计模式不仅帮助我在解决问题时能够更高效地组织代码，还让我深刻认识到面向对象设计的强大之处，尤其是它在增强代码可复用性和可维护性方面的重要作用

### 六、仓库地址

https://github.com/Senwwwwww/Software