好的,我会根据这个需求设计一个符合面向对象设计原则的UML类图,并提供相应的Java代码实现。这个设计将使用策略模式,以便于未来可能的扩展和修改。

首先,这里是UML类图的PlantUML代码:

```plaintext

@startuml

interface DiscountStrategy {

+ calculatePrice(basePrice: double): double

}

class NoDiscount implements DiscountStrategy {

+ calculatePrice(basePrice: double): double

}

class GoldCardDiscount implements DiscountStrategy {

+ calculatePrice(basePrice: double): double

}

class SilverCardDiscount implements DiscountStrategy {

+ calculatePrice(basePrice: double): double

}

class User {

- name: String

- discountStrategy: DiscountStrategy

+ User(name: String, discountStrategy: DiscountStrategy)

+ setDiscountStrategy(discountStrategy: DiscountStrategy): void

+ calculatePrice(basePrice: double): double

}

class HotelBookingSystem {

+ bookRoom(user: User, basePrice: double): void

}

User o-- DiscountStrategy

HotelBookingSystem ..> User

@enduml

```

现在,我们来实现相应的Java代码:

1. 首先是`DiscountStrategy`接口:

```java

public interface DiscountStrategy {

double calculatePrice(double basePrice);

}

```

2. 实现不同的折扣策略:

```java

public class NoDiscount implements DiscountStrategy {

@Override

public double calculatePrice(double basePrice) {

return basePrice;

}

}

public class GoldCardDiscount implements DiscountStrategy {

@Override

public double calculatePrice(double basePrice) {

return basePrice \* 0.8; // 8折

}

}

public class SilverCardDiscount implements DiscountStrategy {

@Override

public double calculatePrice(double basePrice) {

return basePrice \* 0.9; // 9折

}

}

```

3. 实现`User`类:

```java

public class User {

private String name;

private DiscountStrategy discountStrategy;

public User(String name, DiscountStrategy discountStrategy) {

this.name = name;

this.discountStrategy = discountStrategy;

}

public void setDiscountStrategy(DiscountStrategy discountStrategy) {

this.discountStrategy = discountStrategy;

}

public double calculatePrice(double basePrice) {

return discountStrategy.calculatePrice(basePrice);

}

public String getName() {

return name;

}

}

```

4. 实现`HotelBookingSystem`类:

```java

public class HotelBookingSystem {

public void bookRoom(User user, double basePrice) {

double finalPrice = user.calculatePrice(basePrice);

System.out.println(user.getName() + " booked a room. Base price: " + basePrice +

", Final price: " + finalPrice);

}

}

```

5. 最后,我们可以创建一个`Client`类来测试这个系统:

```java

public class Client {

public static void main(String[] args) {

HotelBookingSystem bookingSystem = new HotelBookingSystem();

User normalUser = new User("John", new NoDiscount());

User silverUser = new User("Alice", new SilverCardDiscount());

User goldUser = new User("Bob", new GoldCardDiscount());

double basePrice = 100.0;

bookingSystem.bookRoom(normalUser, basePrice);

bookingSystem.bookRoom(silverUser, basePrice);

bookingSystem.bookRoom(goldUser, basePrice);

// 假设John升级到了金卡

normalUser.setDiscountStrategy(new GoldCardDiscount());

bookingSystem.bookRoom(normalUser, basePrice);

}

}

```

这个设计符合以下面向对象设计原则:

1. 单一职责原则: 每个类都有明确的单一职责。

2. 开放-封闭原则: 系统可以轻松添加新的折扣策略,而不需要修改现有代码。

3. 依赖倒置原则: 高层模块(User)依赖于抽象(DiscountStrategy),而不依赖于具体实现。

4. 里氏替换原则: 所有的DiscountStrategy实现都可以互相替换,不会影响系统的正确性。

5. 接口隔离原则: DiscountStrategy接口只包含了必要的方法。

这个设计允许我们轻松地添加新的用户类型或折扣策略,而不需要修改现有的代码,从而提高了系统的可扩展性和可维护性。