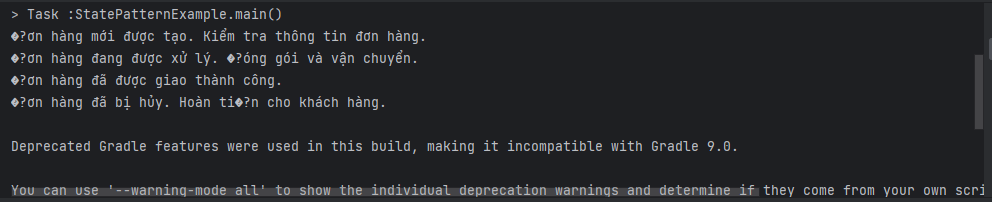
**TUẦN 2**

**Bài 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. Tạo một hệ thống quản lý đơn hàng có các trạng thái như: Mới tạo, Đang xử lý, Đã giao, và Hủy. | | | | |
| Mỗi trạng thái sẽ có các hành vi khác nhau. Ví dụ: |  |  |  |  |
|  | Mới tạo: Kiểm tra thông tin đơn hàng. |  |  |  |
|  | Đang xử lý: Đóng gói và vận chuyển. |  |  |  |
|  | Đã giao: Cập nhật trạng thái đơn hàng là đã giao. |  |  |  |
|  | Hủy: Hủy đơn hàng và hoàn tiền. |  |  |  |
|  | Hãy dùng các Design Pattern: State, Strategy, Decorator viết bằng Java để xử lý mô phỏng trường hợp trên và đưa ra kết luận. | | | |

Design Pattern sử dụng là: State

package org.example;  
  
// Interface trạng thái đơn hàng  
interface OrderState {  
 void handleOrder(OrderContext context);  
}  
  
// Trạng thái Mới tạo  
class NewOrderState implements OrderState {  
 @Override  
 public void handleOrder(OrderContext context) {  
 System.*out*.println("Đơn hàng mới được tạo. Kiểm tra thông tin đơn hàng.");  
 context.setState(new ProcessingOrderState());  
 }  
}  
  
// Trạng thái Đang xử lý  
class ProcessingOrderState implements OrderState {  
 @Override  
 public void handleOrder(OrderContext context) {  
 System.*out*.println("Đơn hàng đang được xử lý. Đóng gói và vận chuyển.");  
 context.setState(new DeliveredOrderState());  
 }  
}  
  
// Trạng thái Đã giao  
class DeliveredOrderState implements OrderState {  
 @Override  
 public void handleOrder(OrderContext context) {  
 System.*out*.println("Đơn hàng đã được giao thành công.");  
 }  
}  
  
// Trạng thái Hủy  
class CancelledOrderState implements OrderState {  
 @Override  
 public void handleOrder(OrderContext context) {  
 System.*out*.println("Đơn hàng đã bị hủy. Hoàn tiền cho khách hàng.");  
 }  
}  
  
// Context quản lý trạng thái đơn hàng  
class OrderContext {  
 private OrderState state;  
  
 public OrderContext() {  
 this.state = new NewOrderState(); // Mặc định là trạng thái mới tạo  
 }  
  
 public void setState(OrderState state) {  
 this.state = state;  
 }  
  
 public void processOrder() {  
 state.handleOrder(this);  
 }  
}  
  
// Main chạy thử  
public class StatePatternExample {  
 public static void main(String[] args) {  
 OrderContext order = new OrderContext();  
  
 order.processOrder(); // Chuyển từ Mới tạo -> Đang xử lý  
 order.processOrder(); // Chuyển từ Đang xử lý -> Đã giao  
 order.processOrder(); // Đã giao (không thay đổi)  
  
 // Trường hợp hủy đơn  
 OrderContext cancelledOrder = new OrderContext();  
 cancelledOrder.setState(new CancelledOrderState());  
 cancelledOrder.processOrder();  
 }  
}



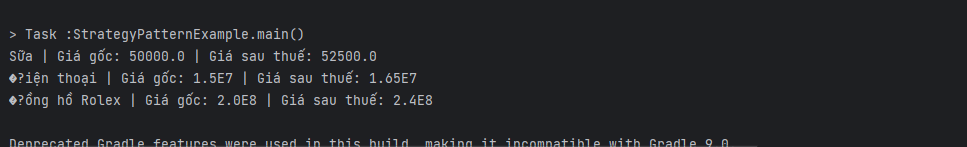
* **Ưu điểm của State Pattern trong bài toán này:**
  + Giúp tách biệt logic của từng trạng thái đơn hàng.
  + Giảm sự phức tạp của code khi xử lý các trạng thái bằng if-else hoặc switch-case.
  + Dễ mở rộng khi thêm trạng thái mới.
* **Hạn chế:**
  + Nếu có quá nhiều trạng thái, có thể tạo ra nhiều lớp dẫn đến khó quản lý.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2. Tính toán thuế cho sản phẩm |  |  |  |  |
|  | Mô tả: |  |  |  |
|  | Giả sử bạn có một hệ thống bán hàng và muốn tính toán thuế cho các sản phẩm khác nhau. |  |  |  |
|  | Các sản phẩm có thể áp dụng các loại thuế khác nhau, ví dụ như thuế tiêu thụ, thuế giá trị gia tăng (VAT), hoặc thuế đặc biệt cho các sản phẩm xa xỉ. | | | |
|  |  |  |  |  |
|  | Hãy dùng các Design Pattern: State, Strategy, Decorator viết bằng Java để xử lý mô phỏng trường hợp trên và đưa ra kết luận. | | | |

Design Pattern sử dụng là: Strategy

**Lý do chọn Strategy Pattern:**

* Hệ thống có nhiều cách tính thuế khác nhau (thuế tiêu thụ, VAT, thuế xa xỉ).
* Dễ dàng mở rộng thêm loại thuế mới mà không ảnh hưởng đến code cũ.
* package org.example;  
    
  import java.util.ArrayList;  
  import java.util.List;  
    
  // Interface chiến lược tính thuế  
  interface TaxStrategy {  
   double calculateTax(double price);  
  }  
    
  // Thuế tiêu thụ (Consumption Tax)  
  class ConsumptionTax implements TaxStrategy {  
   @Override  
   public double calculateTax(double price) {  
   return price \* 0.05; // 5% thuế  
   }  
  }  
    
  // Thuế VAT (Value Added Tax)  
  class VATTax implements TaxStrategy {  
   @Override  
   public double calculateTax(double price) {  
   return price \* 0.1; // 10% VAT  
   }  
  }  
    
  // Thuế xa xỉ (Luxury Tax)  
  class LuxuryTax implements TaxStrategy {  
   @Override  
   public double calculateTax(double price) {  
   return price \* 0.2; // 20% thuế xa xỉ  
   }  
  }  
    
  // Lớp Product chứa thông tin sản phẩm và áp dụng chiến lược thuế  
  class Product {  
   private String name;  
   private double price;  
   private TaxStrategy taxStrategy;  
    
   public Product(String name, double price, TaxStrategy taxStrategy) {  
   this.name = name;  
   this.price = price;  
   this.taxStrategy = taxStrategy;  
   }  
    
   public double getFinalPrice() {  
   return price + taxStrategy.calculateTax(price);  
   }  
    
   public void displayInfo() {  
   System.*out*.println(name + " | Giá gốc: " + price + " | Giá sau thuế: " + getFinalPrice());  
   }  
  }  
    
  // Chạy chương trình  
  public class StrategyPatternExample {  
   public static void main(String[] args) {  
   // Danh sách sản phẩm  
   List<Product> products = new ArrayList<>();  
   products.add(new Product("Sữa", 50000, new ConsumptionTax()));  
   products.add(new Product("Điện thoại", 15000000, new VATTax()));  
   products.add(new Product("Đồng hồ Rolex", 200000000, new LuxuryTax()));  
    
   // Hiển thị thông tin sản phẩm sau khi tính thuế  
   for (Product product : products) {  
   product.displayInfo();  
   }  
   }  
  }



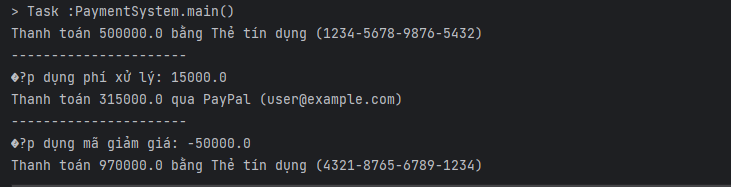
**Kết luận**

* **Ưu điểm:**
  + Có thể dễ dàng thay đổi hoặc thêm mới các chiến lược tính thuế.
  + Code sạch, không cần nhiều if-else.
  + Mở rộng mà không ảnh hưởng đến mã nguồn cũ.
* **Hạn chế:**
  + Nếu có quá nhiều chiến lược thuế, số lượng class có thể tăng lên.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3. Tạo một hệ thống thanh toán, nơi bạn có thể thanh toán bằng các phương thức khác nhau như Thẻ tín dụng, PayPal, và có thể thêm các phương thức thanh toán này với các tính năng bổ sung như Phí xử lý, Mã giảm giá. | | | | |
|  |  |  |  |  |
|  | Hãy dùng các Design Pattern: State, Strategy, Decorator viết bằng Java để xử lý mô phỏng trường hợp trên và đưa ra kết luận. | | | |

Design Pattern sử dụng là: Strategy và Decorator

package org.example;  
  
// Interface chiến lược thanh toán  
interface PaymentStrategy {  
 void pay(double amount);  
}  
  
// Thanh toán bằng Thẻ tín dụng  
class CreditCardPayment implements PaymentStrategy {  
 private String cardNumber;  
  
 public CreditCardPayment(String cardNumber) {  
 this.cardNumber = cardNumber;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.*out*.println("Thanh toán " + amount + " bằng Thẻ tín dụng (" + cardNumber + ")");  
 }  
}  
  
// Thanh toán bằng PayPal  
class PayPalPayment implements PaymentStrategy {  
 private String email;  
  
 public PayPalPayment(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.*out*.println("Thanh toán " + amount + " qua PayPal (" + email + ")");  
 }  
}  
  
// Lớp Payment cơ bản  
class Payment {  
 protected PaymentStrategy strategy;  
 protected double amount;  
  
 public Payment(PaymentStrategy strategy, double amount) {  
 this.strategy = strategy;  
 this.amount = amount;  
 }  
  
 public void processPayment() {  
 strategy.pay(amount);  
 }  
}  
  
// Decorator để thêm phí xử lý  
class ProcessingFeeDecorator extends Payment {  
 private double fee;  
  
 public ProcessingFeeDecorator(Payment basePayment, double fee) {  
 super(basePayment.strategy, basePayment.amount + fee);  
 this.fee = fee;  
 }  
  
 @Override  
 public void processPayment() {  
 System.*out*.println("Áp dụng phí xử lý: " + fee);  
 strategy.pay(amount);  
 }  
}  
  
// Decorator để áp dụng mã giảm giá  
class DiscountDecorator extends Payment {  
 private double discount;  
  
 public DiscountDecorator(Payment basePayment, double discount) {  
 super(basePayment.strategy, basePayment.amount - discount);  
 this.discount = discount;  
 }  
  
 @Override  
 public void processPayment() {  
 System.*out*.println("Áp dụng mã giảm giá: -" + discount);  
 strategy.pay(amount);  
 }  
}  
  
// Chạy chương trình  
public class PaymentSystem {  
 public static void main(String[] args) {  
 // Thanh toán bằng thẻ tín dụng, không có phí xử lý  
 Payment payment1 = new Payment(new CreditCardPayment("1234-5678-9876-5432"), 500000);  
 payment1.processPayment();  
  
 System.*out*.println("----------------------");  
  
 // Thanh toán qua PayPal + Phí xử lý  
 Payment payment2 = new ProcessingFeeDecorator(  
 new Payment(new PayPalPayment("user@example.com"), 300000), 15000);  
 payment2.processPayment();  
  
 System.*out*.println("----------------------");  
  
 // Thanh toán bằng thẻ tín dụng + Phí xử lý + Giảm giá  
 Payment payment3 = new DiscountDecorator(  
 new ProcessingFeeDecorator(  
 new Payment(new CreditCardPayment("4321-8765-6789-1234"), 1000000), 20000), 50000);  
 payment3.processPayment();  
 }  
}



**Strategy Pattern** giúp thay đổi phương thức thanh toán dễ dàng.

**Decorator Pattern** cho phép thêm phí xử lý hoặc mã giảm giá mà không làm thay đổi code gốc.

**Dễ mở rộng:** Có thể thêm **chuyển khoản ngân hàng**, **tiền điện tử**, hoặc các ưu đãi khác mà không sửa code cũ.