**REPORT FOR AIMS CAPSTONE PROJECT**

*(for all use cases related to Place order)*

The Single Responsibility Principle (SRP)

Student ID : 20215198

Student name : Dương Hoàng Hải

**1. Class ViewCartController**

**Violation of SRP**

The ViewCartController class has multiple responsibilities:

* **Managing the cart:**
  + Retrieving all products in the cart (getAllProductsInCart)
  + Updating the cart (updateCart)
* **Checking product inventory:**
  + Checking product stock quantity (checkQtyInStock)
* **Calculating tax:**
  + Getting tax information (getTax)

**Cohesion**

**Cohesion** refers to how closely related and focused the responsibilities of a class are. In ViewCartController, the methods are not entirely related:

* **Cart management methods:**
  + getAllProductsInCart()
  + updateCart()
* **Inventory checking method:**
  + checkQtyInStock()
* **Tax calculation method:**
  + getTax()

These methods serve different purposes, resulting in low cohesion for the ViewCartController class.

To adhere to SRP and increase cohesion, we should split the different responsibilities into separate controllers.

**Class CartController**

@RestController

@RequestMapping("/cart")

public class CartController {

private final CartService cartService;

@Autowired

public CartController(CartService cartService) {

this.cartService = cartService;

}

@GetMapping

public ResponseEntity<List<CartProduct>> getAllProductsInCart() {

List<CartProduct> cartProducts = cartService.getAllProductsInCart(1L);

if (cartProducts == null) {

return ResponseEntity.notFound().build();

}

return ResponseEntity.ok(cartProducts);

}

@PostMapping

public ResponseEntity<UpdateCartResponse> updateCart(@RequestBody Map<String, Object> request) {

Long productId = Long.valueOf(request.get("product\_id").toString());

Integer qty = Integer.valueOf(request.get("qty").toString());

try {

List<CartProduct> cart = cartService.updateCart(1L, productId, qty);

return ResponseEntity.ok(new UpdateCartResponse("Cart updated successfully", cart));

} catch (Exception e) {

return ResponseEntity.notFound().build();

}

}

}

**Class InventoryController**

@RestController

@RequestMapping("/inventory")

public class InventoryController {

private final ProductService productService;

@Autowired

public InventoryController(ProductService productService) {

this.productService = productService;

}

@GetMapping("/check")

public ResponseEntity<StockAvailabilityResponse> checkQtyInStock(@RequestParam("product\_id") Long productId,

@RequestParam("qty") Integer qty) {

try {

boolean isAvailable = productService.checkInventory(productId, qty);

return ResponseEntity.ok(new StockAvailabilityResponse(productId, qty, isAvailable));

} catch (Exception e) {

return ResponseEntity.notFound().build();

}

}

}

**Class TaxController**

@RestController

@RequestMapping("/tax")

public class TaxController {

@GetMapping

public ResponseEntity<TaxResponse> getTax() {

return ResponseEntity.ok(new TaxResponse(10));

}

}

**Adherence to SRP**

* CartController is responsible only for managing the cart, including retrieving and updating products in the cart.
* InventoryController is responsible only for checking product inventory.
* TaxController is responsible only for retrieving tax information.

**Cohesion**

* CartController: The methods in this class are all related to managing the cart, leading to high cohesion.
* InventoryController: The methods in this class are all related to checking product inventory, leading to high cohesion.
* TaxController: This class has a single method related to tax, leading to high cohesion.

**2. Class PaymentController**

**Violation of SRP**

The PaymentController class violates the Single Responsibility Principle (SRP) because it handles multiple responsibilities:

* **Handling HTTP requests:**
  + Processes GET and POST requests related to payments.
* **Managing invoices:**
  + Creates an invoice (payOrder).
  + Stores the invoice (makePayment).
* **Integrating with the payment subsystem:**
  + Generates a payment URL (generateURL).
  + Retrieves payment transaction information (makePayment).

**Cohesion**

The PaymentController class has low cohesion because its methods are not closely related and serve different purposes:

* **HTTP request handling:**
  + getInvoiceDetail()
  + makePayment()
  + generateURL()
* **Invoice management:**
  + payOrder()
* **Payment subsystem integration:**
  + payment.getPaymentTransaction(res)
  + payment.generateURL(invoice.getAmount(), "Payment")

To adhere to SRP and increase cohesion, we should split the different responsibilities into separate classes.

**Class PaymentController**

@RestController

@RequestMapping("/payment")

public class PaymentController {

private final PaymentService paymentService;

@Autowired

public PaymentController(PaymentService paymentService) {

this.paymentService = paymentService;

}

@GetMapping("/invoice")

public ResponseEntity<InvoiceDetailResponse> getInvoiceDetail() {

Invoice invoice = paymentService.getCurrentInvoice();

return ResponseEntity.ok(new InvoiceDetailResponse(invoice.getOrder(), paymentService.getCartService()));

}

@PostMapping("/result")

public ResponseEntity<Void> makePayment(@RequestBody Map<String, String> res) throws IOException, SQLException {

paymentService.processPaymentResult(res);

return ResponseEntity.ok().build();

}

@GetMapping("/VNPayURL")

public PaymentURLResponse generateURL() throws IOException {

return paymentService.generatePaymentURL();

}

}

**Class PaymentService**

@Service

public class PaymentService {

private final IPaymentSubsystem payment;

private Invoice currentInvoice;

private final InvoiceService invoiceService;

private final CartService cartService;

@Autowired

public PaymentService(InvoiceService invoiceService, CartService cartService) {

this.payment = new PaymentSubsystem(new VNPaySubsystemController());

this.invoiceService = invoiceService;

this.cartService = cartService;

}

public Invoice getCurrentInvoice() {

return currentInvoice;

}

public void createInvoice(Order order) {

this.currentInvoice = new Invoice(order);

}

public void processPaymentResult(Map<String, String> res) throws IOException, SQLException {

PaymentTransaction transaction = payment.getPaymentTransaction(res);

currentInvoice.setPaymentTransaction(transaction);

invoiceService.save(currentInvoice);

}

public PaymentURLResponse generatePaymentURL() throws IOException {

return new PaymentURLResponse(payment.generateURL(currentInvoice.getAmount(), "Payment"));

}

public CartService getCartService() {

return cartService;

}

}

**Adherence to SRP**

* PaymentController now only handles HTTP requests related to payments.
* PaymentService handles the business logic related to payments, including invoice creation, payment processing, and generating payment URLs.

**Cohesion**

* PaymentController: The methods in this class are all related to handling HTTP requests for payments, leading to high cohesion.
* PaymentService: The methods in this class are all related to payment processing logic, leading to high cohesion.

**3. Class PlaceOrderController**

**Violation of SRP**

The PlaceOrderController class violates the Single Responsibility Principle (SRP) because it handles multiple responsibilities:

* **Managing cart delivery details:**
  + Retrieving cart delivery information (getCartDelivery)
  + Checking rush order availability (checkRushOrder)
  + Submitting the delivery form (submitDeliveryForm)
* **Calculating shipping fees:**
  + Calculating normal shipping fees (calculateNormalShippingFee)
  + Calculating rush shipping fees (calculateRushShippingFee)
* **Handling payment integration:**
  + Interacting with PaymentController to process payment (submitDeliveryForm)

**Cohesion**

The PlaceOrderController class has low cohesion because its methods are not closely related and serve different purposes:

* **Cart delivery management:**
  + getCartDelivery()
  + checkRushOrder()
  + submitDeliveryForm()
* **Shipping fee calculations:**
  + calculateNormalShippingFee()
  + calculateRushShippingFee()
* **Payment integration:**
  + Interacting with PaymentController to initiate payment (submitDeliveryForm)

To adhere to SRP and increase cohesion, we should split the different responsibilities into separate classes.

**Class DeliveryController**

@RestController

@RequestMapping("/cart/delivery")

public class DeliveryController {

private final CartService cartService;

private final PaymentController paymentController;

@Autowired

public DeliveryController(PaymentController paymentController,CartService cartService) {

this.paymentController = paymentController;

this.cartService = cartService;

}

@GetMapping

public ResponseEntity<CartProductResponse> getCartDelivery(@RequestParam("cartId") Long cartId) {

try {

List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);

CartProductResponse response = CartProductResponse.fromCartProducts(cartProducts);

return ResponseEntity.ok(response);

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.notFound().build();

}

}

@PostMapping("/checkRushOrder")

public ResponseEntity<RushDeliveryCheckResponse> checkRushOrder(@RequestBody Map<String, Object> request) {

try {

Long cartId = Long.valueOf(request.get("cartId").toString());

Integer province = request.containsKey("province") ? Integer.valueOf(request.get("province").toString()) : null;

Boolean isRushDelivery = Boolean.valueOf(request.get("isRushDelivery").toString());

List<CartProduct> cartProducts = cartService.getAllProductsInCart(cartId);

DeliveryService deliveryService = new DeliveryService();

RushDeliveryCheckResponse response = deliveryService.checkRushOrder(cartProducts, province, isRushDelivery);

return ResponseEntity.ok(response);

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.notFound().build();

}

}

@PostMapping("/submit")

public ResponseEntity<String> submitDeliveryForm(@RequestBody Map<String, Object> request) {

try {

DeliveryService deliveryService = new DeliveryService();

Order order = deliveryService.createOrderFromRequest(request, cartService);

paymentController.payOrder(order);

return ResponseEntity.ok("Order created successfully");

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(404).body("Failed to create order");

}

}

}

**Class DeliveryService**

@Service

public class DeliveryService {

public RushDeliveryCheckResponse checkRushOrder(List<CartProduct> cartProducts, Integer province, Boolean isRushDelivery) {

if (province == null) {

return new RushDeliveryCheckResponse(0, 0, false);

} else if (!isRushDelivery || province != 1) {

double normalShippingFee = calculateNormalShippingFee(cartProducts, province);

return new RushDeliveryCheckResponse(normalShippingFee, 0, false);

} else {

List<CartProduct> rushDeliveryProducts = getRushDeliveryProducts(cartProducts);

if (rushDeliveryProducts.isEmpty()) {

double normalShippingFee = calculateNormalShippingFee(cartProducts, province);

return new RushDeliveryCheckResponse(normalShippingFee, 0, false);

} else {

List<CartProduct> normalDeliveryProducts = getNonRushDeliveryProducts(cartProducts);

double normalShippingFee = calculateNormalShippingFee(normalDeliveryProducts, province);

double rushShippingFee = calculateRushShippingFee(rushDeliveryProducts, province);

return new RushDeliveryCheckResponse(normalShippingFee, rushShippingFee, true);

}

}

}

public Order createOrderFromRequest(Map<String, Object> request, CartService cartService) throws Exception {

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

Long cartId = Long.valueOf(request.get("cartId").toString());

Map<String, Object> deliveryFormDTO = (Map<String, Object>) request.get("DeliveryFormDTO");

String name = deliveryFormDTO.get("name").toString();

String phone = deliveryFormDTO.get("phone").toString();

String email = deliveryFormDTO.get("email").toString();

String address = deliveryFormDTO.get("address").toString();

Long province = Long.valueOf(deliveryFormDTO.get("province").toString());

String instructions = deliveryFormDTO.get("note").toString();

LocalDate date = LocalDate.parse(deliveryFormDTO.get("date").toString(), formatter);

Boolean isRushDelivery = Boolean.valueOf(deliveryFormDTO.get("isRushDelivery").toString());

DeliveryInfo deliveryInfo = new DeliveryInfo(name, phone, email, province, instructions, address, date, isRushDelivery);

if (!deliveryInfo.isValid()) {

throw new Exception("Invalid delivery information");

}

Cart cart = cartService.findById(cartId);

double normalShippingFees = calculateNormalShippingFee(cart.getProducts(), province.intValue());

double rushShippingFees = isRushDelivery ? calculateRushShippingFee(cart.getProducts(), province.intValue()) : 0.0;

return new Order(cart, normalShippingFees, rushShippingFees, deliveryInfo);

}

public double calculateNormalShippingFee(List<CartProduct> cartProducts, Integer province) {

double totalPrice = calculateTotalPrice(cartProducts);

double maxWeight = findProductWithMaxWeight(cartProducts);

int baseRate;

int additionalFeePerHalfKg = 2500; // Additional fee for every subsequent 0.5kg

double shippingFee;

// if the province is Hanoi or Ho Chi Minh city

if (province == 1 || province == 79) {

baseRate = 22000; // Initial price for the first 3kg

if (maxWeight <= 3) {

shippingFee = baseRate;

} else {

shippingFee = baseRate + Math.ceil((maxWeight - 3) / 0.5) \* additionalFeePerHalfKg;

}

} else {

baseRate = 30000; // Initial price for the first 0.5kg

if (maxWeight <= 0.5) {

shippingFee = baseRate;

} else {

shippingFee = baseRate + Math.ceil((maxWeight - 0.5) / 0.5) \* additionalFeePerHalfKg;

}

}

// Apply free shipping if total price exceeds 100,000 VND, up to a maximum of

// 25,000 VND

if (totalPrice > 100000) {

shippingFee = Math.max(shippingFee - 25000, 0);

}

return shippingFee;

}

public double calculateRushShippingFee(List<CartProduct> cartProducts, Integer province) {

int baseRate;

int additionalFeePerHalfKg = 2500; // Additional fee for every subsequent 0.5kg

double maxWeight = findProductWithMaxWeight(cartProducts);

double shippingFee;

if (province == 1 || province == 79) {

baseRate = 22000; // Initial price for the first 3kg

if (maxWeight <= 3) {

shippingFee = baseRate;

} else {

shippingFee = baseRate + Math.ceil((maxWeight - 3) / 0.5) \* additionalFeePerHalfKg;

}

} else {

baseRate = 30000; // Initial price for the first 0.5kg

if (maxWeight <= 0.5) {

shippingFee = baseRate;

} else {

shippingFee = baseRate + Math.ceil((maxWeight - 0.5) / 0.5) \* additionalFeePerHalfKg;

}

}

shippingFee += cartProducts.size() \* 10000;

return shippingFee;

}

public double findProductWithMaxWeight(List<CartProduct> cartProducts) {

return cartProducts.stream()

.mapToDouble(cartProduct -> cartProduct.getProduct().getWeight() \* cartProduct.getQty()).max()

.orElse(0.0);

}

public double calculateTotalPrice(List<CartProduct> cartProducts) {

return cartProducts.stream()

.mapToDouble(cartProduct -> cartProduct.getProduct().getPrice() \* cartProduct.getQty()).sum();

}

public List<CartProduct> getRushDeliveryProducts(List<CartProduct> cartProducts) {

return cartProducts.stream().filter(cartProduct -> cartProduct.getProduct().isRushOrderEligible())

.collect(Collectors.toList());

}

public List<CartProduct> getNonRushDeliveryProducts(List<CartProduct> cartProducts) {

return cartProducts.stream().filter(cartProduct -> !cartProduct.getProduct().isRushOrderEligible())

.collect(Collectors.toList());

}

}

**Adherence to SRP**

* DeliveryController now only handles HTTP requests related to cart delivery and order submission.
* DeliveryService handles the business logic related to delivery, including checking rush orders and calculating shipping fees.

**Cohesion**

* DeliveryController: The methods in this class are all related to handling HTTP requests for cart delivery and order submission, leading to high cohesion.
* DeliveryService: The methods in this class are all related to delivery processing logic, leading to high cohesion.