1. **Depency Inversion:**The Dependency Inversion Principle (DIP) is one of the SOLID principles of object-oriented design. It states that high-level modules should not depend on low-level modules, but both should depend on abstractions. Additionally, abstractions should not depend on details, but details should depend on abstractions.
2. **PlaceOrderController.java**

* Areas for Improvement:
* Direct Instantiation: The PlaceOrderController directly interacts with concrete implementations of Cart, Order, and DeliveryInfo. While this is not a strict violation of DIP, it can be improved by depending on interfaces or abstract classes instead of concrete implementations. This would make the code more flexible and easier to test.
* Hardcoded Logic: Business logic like calculateNormalShippingFee, calculateRushShippingFee, and validation logic in validatePhoneNumber, validateName, etc., are directly within the controller. This could be moved to service classes to make the controller more focused on handling HTTP requests and responses.
* Recommendations:
* Use Interfaces: Define interfaces for Cart, Order, and DeliveryInfo, and let the controller depend on these interfaces. This makes your code more modular and easier to extend or modify.
* Move Business Logic to Services: Move the shipping fee calculation and validation logic to their respective service classes. This helps in keeping the controller lean and focused on its primary responsibility, which is handling HTTP requests and responses.
* Abstract the Payment Controller: Ensure PaymentController is an abstraction (interface or abstract class) to allow flexibility in changing the payment processing logic without modifying the controller.
* Refactored Examples:

@RestController

@RequestMapping

public class PlaceOrderController {

private final PaymentService paymentService;

private final CartService cartService;

private final ShippingService shippingService;

private final OrderService orderService;

private Double normalShippingFees = 0.0;

private Double rushShippingFees = 0.0;

@Autowired

public PlaceOrderController(PaymentService paymentService, CartService cartService, ShippingService shippingService, OrderService orderService) {

this.paymentService = paymentService;

this.cartService = cartService;

this.shippingService = shippingService;

this.orderService = orderService;

}

@PostMapping("/cart/delivery/submit")

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

try {

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()) {

return ResponseEntity.status(404).body("Invalid delivery information");

}

Cart cart = cartService.findById(cartId);

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

this.normalShippingFees = shippingService.calculateNormalShippingFee(cartProducts, province);

if (isRushDelivery) {

List<CartProduct> rushDeliveryProducts = shippingService.getRushDeliveryProducts(cartProducts);

if (rushDeliveryProducts != null && !rushDeliveryProducts.isEmpty()) {

this.rushShippingFees = shippingService.calculateRushShippingFee(rushDeliveryProducts, province);

}

}

Order order = new Order(cart, deliveryInfo, normalShippingFees + rushShippingFees);

orderService.save(order);

paymentService.payOrder(order);

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

} catch (Exception e) {

e.printStackTrace();

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

}

}

// Other methods

}