The Interface Segregation Principle (ISP) is another fundamental principle in the SOLID principles of object-oriented design. It states that no client should be forced to depend on methods it does not use. This principle encourages the creation of smaller, more specific interfaces rather than larger, more general ones.

To design the **PlaceOrderController** according to the ISP, we should break down large interfaces into smaller, more focused ones. This ensures that each interface contains only the methods relevant to a particular client.

### Refactored PlaceOrderController with ISP

Here’s how we can refactor the **PlaceOrderController** to adhere to the ISP:

1. **Define Smaller Interfaces:** Split large service interfaces into smaller, more specific ones.
2. **Use Specific Interfaces in the Controller:** Modify the controller to depend on these smaller interfaces.

### Step 1: Define Smaller Interfaces

First, we define smaller, more focused interfaces:

public interface CartRetrievalService {

Cart findById(Long cartId);

List<CartProduct> getAllProductsInCart(Long cartId);

}

public interface ShippingFeeService {

double calculateNormalShippingFee(List<CartProduct> cartProducts, Long province);

List<CartProduct> getRushDeliveryProducts(List<CartProduct> cartProducts);

double calculateRushShippingFee(List<CartProduct> rushDeliveryProducts, Long province);

}

public interface PaymentProcessingService {

void payOrder(Order order);

}

public interface OrderManagementService {

void save(Order order);

}

public interface DeliveryInfoValidationService {

boolean isValid();

}

### Step 2: Implement these interfaces in the respective service classes

The existing service classes should implement these new interfaces. For example:

@Service

public class CartService implements CartRetrievalService {

// Implement methods

}

@Service

public class ShippingService implements ShippingFeeService {

// Implement methods

}

@Service

public class PaymentService implements PaymentProcessingService {

// Implement methods

}

@Service

public class OrderService implements OrderManagementService {

// Implement methods

}

public class DeliveryInfo implements DeliveryInfoValidationService {

// Implement methods

}

### Step 3: Use Specific Interfaces in the Controller

Modify the **PlaceOrderController** to depend on these smaller interfaces:

@RestController

@RequestMapping

public class PlaceOrderController {

private final PaymentProcessingService paymentProcessingService;

private final CartRetrievalService cartRetrievalService;

private final ShippingFeeService shippingFeeService;

private final OrderManagementService orderManagementService;

@Autowired

public PlaceOrderController(PaymentProcessingService paymentProcessingService, CartRetrievalService cartRetrievalService, ShippingFeeService shippingFeeService, OrderManagementService orderManagementService) {

this.paymentProcessingService = paymentProcessingService;

this.cartRetrievalService = cartRetrievalService;

this.shippingFeeService = shippingFeeService;

this.orderManagementService = orderManagementService;

}

@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 = cartRetrievalService.findById(cartId);

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

double normalShippingFees = shippingFeeService.calculateNormalShippingFee(cartProducts, province);

double rushShippingFees = 0.0;

if (isRushDelivery) {

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

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

rushShippingFees = shippingFeeService.calculateRushShippingFee(rushDeliveryProducts, province);

}

}

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

orderManagementService.save(order);

paymentProcessingService.payOrder(order);

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

} catch (Exception e) {

e.printStackTrace();

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

}

}

// Other methods

}

### Explanation:

1. **Specific Interfaces:** The controller now depends on smaller, specific interfaces (**CartRetrievalService**, **ShippingFeeService**, **PaymentProcessingService**, and **OrderManagementService**).
2. **Focused Responsibilities:** Each service interface has a clear responsibility, ensuring that clients (like the **PlaceOrderController**) are not forced to depend on methods they do not use.
3. **Adherence to ISP:** This refactoring adheres to the ISP, making the codebase more modular, maintainable, and testable.

By following the ISP, we ensure that the **PlaceOrderController** is only dependent on the necessary methods, which simplifies the code and enhances maintainability.