HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
School of Information and Communications Technology

Software Requirement Specification

AIMS – An Internet Media Store ITSS Software Development

Nhóm 20
Design Pattern

Họ và tên	MSSV	Vai trò
Nguyễn Duy Tấn	20215478	Nhóm trưởng
Nguyễn Văn Tấn	20215479	Thành viên
Lù Mạnh Thắng	20194167	Thành viên
Lưu Trọng Tấn	20215477	Thành viên
thanongsith thavisack	20180288	Thành viên

Hanoi, January 2025

1. Vấn đề 1: thêm sản phẩm mới AudioBook

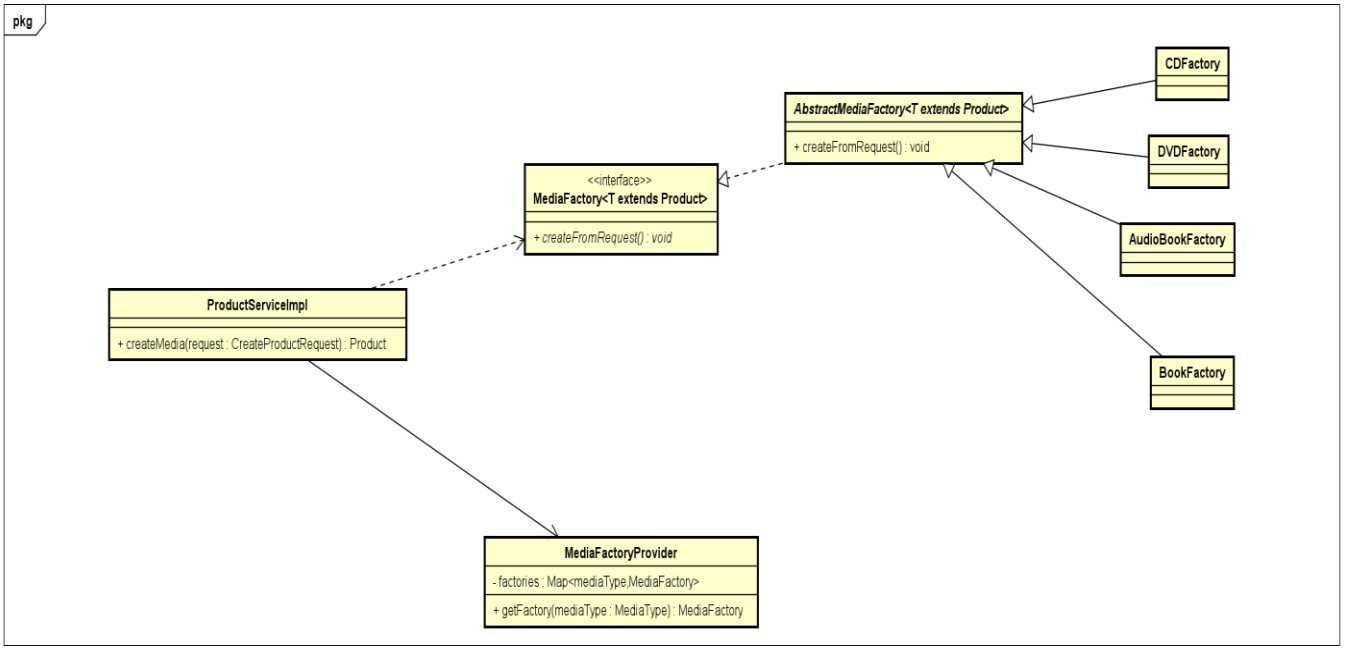
Đối với yêu cầu 1: Thêm mặt hàng Media mới: AudioBook
Thông tin về loại mặt hàng mới như sau:
AudioBook
+ String author, ví dụ: Paulo Coelho
+ String format, ví dụ: mp3
+ String language, ví dụ: English
+ String accent, ví dụ: Male - North America
+ int lengthInMinutes, ví dụ: 226

Vấn đề

Cơ sở dữ liệu của nhóm đang sử dụng MongoDB lưu tất cả các loại sản phẩm vào trong Document Product. Việc thêm một sản phẩm mới trong cơ sở dữ liệu sẽ gây ra control coupling, đồng thời vi phạm tính Open-Close Principle trong SOLID minh họa trong đoạn code sau

```
if(request.getMediaType() == MediaType.BOOK) {
    Product Media = productService.createBook(request);
} else if(request.getMediaType() == MediaType.CD) {
    Product Media = productService.createCD(request);
} else if(request.getMediaType() == MediaType.AUDIO_BOOK) {
    Product Media = productService.createAudioBook(request);    You,
}
```

Để giải quyết vấn đề trên Nhóm đã áp dụng factory Method như sau.



Triển khai cụ thể

```
    @PostMapping("/create")
    public ResponseEntity<AIMSResponse<Object>> createMedia(@Valid @RequestBody CreateProductRequest request) {
        Product media = productService.createMedia(request);
        return ResponseUtil.success201Response( message: "Create media successfully", media);
    }   Tan-Tien-Le, 12/29/2024 9:40 PM • feat: factory pattern method for creating audiobook
}
```

Đối với mỗi request để tạo loại sản phẩm mới, mỗi loại sản phẩm có các thuộc tính riêng được gửi lên sử dụng extend để kế thừa các thuộc tính chung.

```java
public class AudioBookRequest extends CreateProductRequest {  no usages        Tan-Tien-Le, 12/29/2024 9:40 PM • feat: f
    @NotBlank(message = "Author is required")  no usages
    private String author;

    @NotBlank(message = "Format is required")  no usages
    private String format;

    @NotBlank(message = "Language is required")  no usages
    private String language;

    private String accent;  no usages

    @Min(value = 1, message = "Length must be positive")  no usages
    private int lengthInMinutes;

    AudioBookRequest(String title, int sellPrice, int quantity, String imageURL, boolean rushDeliverySupport, MediaType
        super(title, sellPrice, quantity, imageURL, rushDeliverySupport, mediaType);
    }
}
```

Phương thức createMedia được gọi có triển khai như sau

```java
    @Override  1 usage
    public Product createMedia(CreateProductRequest request) {    Tan-Tien-Le, 12/29/202
        // Get appropriate factory
        MediaFactory factory = factoryProvider.getFactory(request.getMediaType());

        // Create media object
        Product media = factory.createFromRequest(request);

        // Save to repository
        return productRepository.save(media);
    }
```

Dựa vào type trong product sẽ lấy được các factory như AudioFactory, AudioFactory sẽ sử dụng phương thức createFromRequest() được kế thừa từ AbstractMediaFactory , và trong lớp AbstractMediaFactory phương thức createFromRequest() đã được implement. Tạo 1 sản phẩm mới từ request dựa trên request được gửi.

Triển khai của các factory như sau

```java
public interface MediaFactory<T extends Product> {  8 usages  5 implementations
    T createMedia();  no usages  1 implementation        Tan-Tien-Le, 12/29/2024 9:40 PM • feat: factor
    T createFromRequest(CreateProductRequest request);  1 usage  1 implementation
}
```

```java
@Component  4 inheritors
public abstract class AbstractMediaFactory<T extends Product> implements MediaFactory<T> {

    @Override  no usages
    public T createMedia() {
        T media = createEmptyMedia();    Tan-Tien-Le, 12/29/2024 9:40 PM • feat: factory pattern method for creati
        initializeDefaultValues(media);
        return media;
    }

    @Override  1 usage
    public T createFromRequest(CreateProductRequest request) {
        T media = createEmptyMedia();
        mapRequestToMedia(media, request);
        return media;
    }

    protected abstract T createEmptyMedia();  2 usages  4 implementations
    protected abstract void initializeDefaultValues(T media);  1 usage  4 implementations
    protected abstract void mapRequestToMedia(T media, CreateProductRequest request);  1 usage  4 implementations
}
```

```java
@Component
public class AudioBookFactory extends AbstractMediaFactory<AudioBook> {

    @Override  2 usages
    protected AudioBook createEmptyMedia() { return new AudioBook(); }

    @Override  1 usage
    protected void initializeDefaultValues(AudioBook audioBook) {
        audioBook.setType(MediaType.AUDIO_BOOK.toString());
        audioBook.setFormat("MP3");
        // Set other default values
    }



    @Override  1 usage
    protected void mapRequestToMedia(AudioBook audioBook, CreateProductRequest request) {
//        if (!(request instanceof AudioBookRequest)) {
//            throw new IllegalArgumentException("Invalid request type");
//        }
//        AudioBookRequest audioBookRequest = (AudioBookRequest) request;
//
//        audioBook.setTitle(audioBookRequest.getTitle());
//        audioBook.setAuthor(audioBookRequest.getAuthor());
//        audioBook.setFormat(audioBookRequest.getFormat());
//        audioBook.setLanguage(audioBookRequest.getLanguage());
//        audioBook.setAccent(audioBookRequest.getAccent());
//        audioBook.setLengthInMinutes(audioBookRequest.getLengthInMinutes());    Tan-Tien-Le, 12/29/2024 9:40 PM • fe
    }
}
```

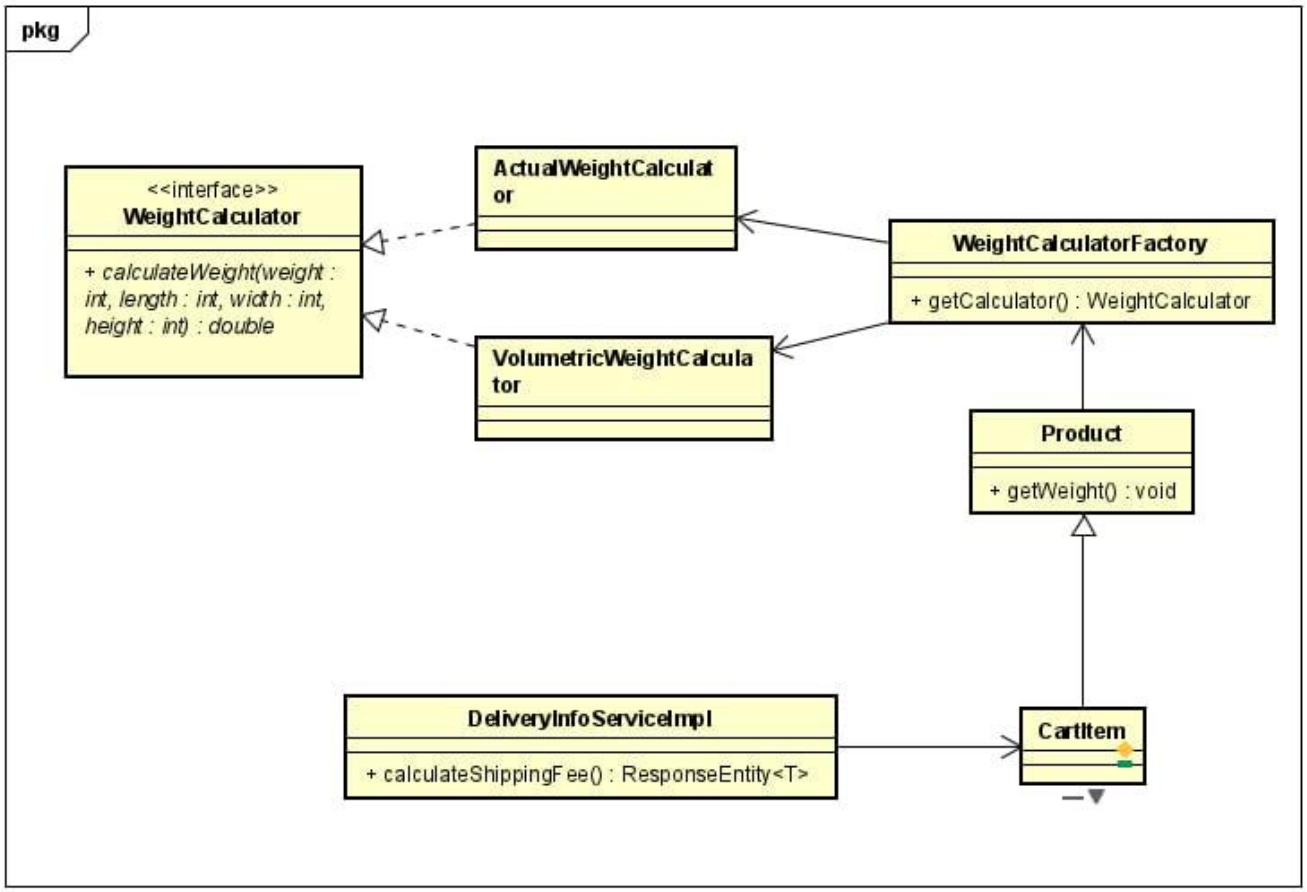
2. Vấn đề 2: Thay đổi cách tính phí vận chuyển

Vấn đề:

Trong hệ thống hiện tại, phí vận chuyển được tính dựa trên khối lượng thực tế của kiện hàng. Tuy nhiên, yêu cầu mới đòi hỏi phải xem xét thêm khối lượng quy đổi (dựa trên kích thước) và tính phí dựa trên giá trị lớn hơn giữa hai khối lượng. Việc tích hợp logic này vào các phương thức tính phí có thể gây ra "control coupling", làm hệ thống khó bảo trì.

Giải pháp:

Nhóm đã áp dụng Factory Method để tách biệt việc quyết định cách tính khối lượng và logic tính phí. Factory Method được sử dụng để tạo các đối tượng thực hiện tính khối lượng phù hợp dựa trên dữ liệu đầu vào.



Triển khai:

a. Factory Method để chọn chiến lược tính khối lượng:

Phương pháp này quyết định sử dụng cách tính khối lượng dựa trên trọng lượng thực tế hoặc khối lượng quy đổi bằng cách tạo các đối tượng chiến lược.

```java
// Interface chung cho các chiến lược tính khối lượng
public interface WeightCalculator {
    double calculateWeight(double weight, double length, double width, double height);
}

// Tính khối lượng thực tế
public static class ActualWeightCalculator implements WeightCalculator {
    @Override
    public double calculateWeight(double weight, double length, double width, double height) {
        return weight;
    }
}

// Tính khối lượng quy đổi
public static class VolumetricWeightCalculator implements WeightCalculator {
    @Override
    public double calculateWeight(double weight, double length, double width, double height) {
        return (length * width * height) / 6000;
    }
}

// Factory Method để chọn chiến lược tính khối lượng
public class WeightCalculatorFactory {
    public WeightCalculator getCalculator(double weight, double length, double width, double height) {
        double volumetricWeight = (length * width * height) / 6000;
        if (weight ≥ volumetricWeight) {
            return new ActualWeightCalculator(); // Sử dụng khối lượng thực tế
        } else {
            return new VolumetricWeightCalculator(); // Sử dụng khối lượng quy đổi
        }
    }
}
```

b. Hàm getWeight trong lớp Product:

Để đơn giản hóa việc truy xuất khối lượng, phương thức getWeight sử dụng WeightCalculatorFactory để tạo chiến lược và tính toán khối lượng phù hợp.

```java
public int getWeight() {
    ShippingFactory.WeightCalculator calculator = new ShippingFactory.WeightCalculatorFactory()
            .getCalculator(weight, length, width, height);
    return (int) calculator.calculateWeight(weight, length, width, height);
}
```

c. Hàm calculateShippingFee:

Hàm này tính phí vận chuyển dựa trên các tham số đầu vào như ID giỏ hàng, tỉnh thành, và yêu cầu giao gấp. Logic bao gồm:

- Xác định khối lượng tổng: Sử dụng phương thức getWeight() cho từng sản phẩm trong giỏ hàng.
- Tính phí giao hàng nội tỉnh hoặc ngoại tỉnh: Phí giao hàng được tính theo quy tắc có sẵn, có tính đến khối lượng tổng và các ngưỡng phụ phí.
- Tính phí giao gấp: Thêm phí giao gấp dựa trên số lượng sản phẩm.
- Miễn phí vận chuyển: Giảm trừ phí giao hàng nếu tổng giá trị giỏ hàng đạt ngưỡng quy định.

```java
@Override
public ResponseEntity<AIMSResponse<Object>> calculateShippingFee(String cartId, String province, boolean isRushDelivery) {
    Cart cart = cartService.getCart(cartId);
    List<CartItem> listItems = cart.getListCartItem();
    double totalWeight = listItems.stream() Stream<CartItem>
            .mapToInt(CartItem::getWeight) IntStream
            .sum();

    int baseFee;
    if (isInnerProvince(province)) {
        baseFee = INNER_CITY_BASE_FEE;
        if (totalWeight > 3) {
            baseFee += (int) (Math.ceil((totalWeight - 3) / 0.5) * ADDITIONAL_FEE);
        }
    } else {
        baseFee = OUTER_CITY_BASE_FEE;
        if (totalWeight > 0.5) {
            baseFee += (int) (Math.ceil((totalWeight - 0.5) / 0.5) * ADDITIONAL_FEE);
        }
    }

    if (isRushDelivery) {
        baseFee += RUSH_ORDER_FEE * listItems.size();
    }

    if (!isRushDelivery && cart.getTotalPrice() > FREE_SHIPPING_THRESHOLD) {
        baseFee -= Math.min(baseFee, MAX_FREE_SHIPPING);
    }

    baseFee =  Math.max(baseFee, 0);
    return ResponseUtil.success200Response( message: "Shipping fee calculated successfully", baseFee);
}
```

Kết quả:

Thiết kế này giúp tích hợp logic tính khối lượng và phí vận chuyển một cách tách biệt và linh hoạt. Việc áp dụng Factory Method đảm bảo rằng việc tạo và chọn chiến lược tính toán khối lượng được thực hiện chính xác và dễ bảo trì.

3. Vấn đề 3 Thêm phương thức thanh toán mới

Đối với vấn đề

3. Thêm phương thức thanh toán mới: Thẻ nội địa (Domestic Card)

Khách hàng có thể lựa chọn phương thức thanh toán thẻ tín dụng (credit card) hoặc thẻ nội địa (domestic card). Thông tin về loại thẻ thanh toán mới như sau:

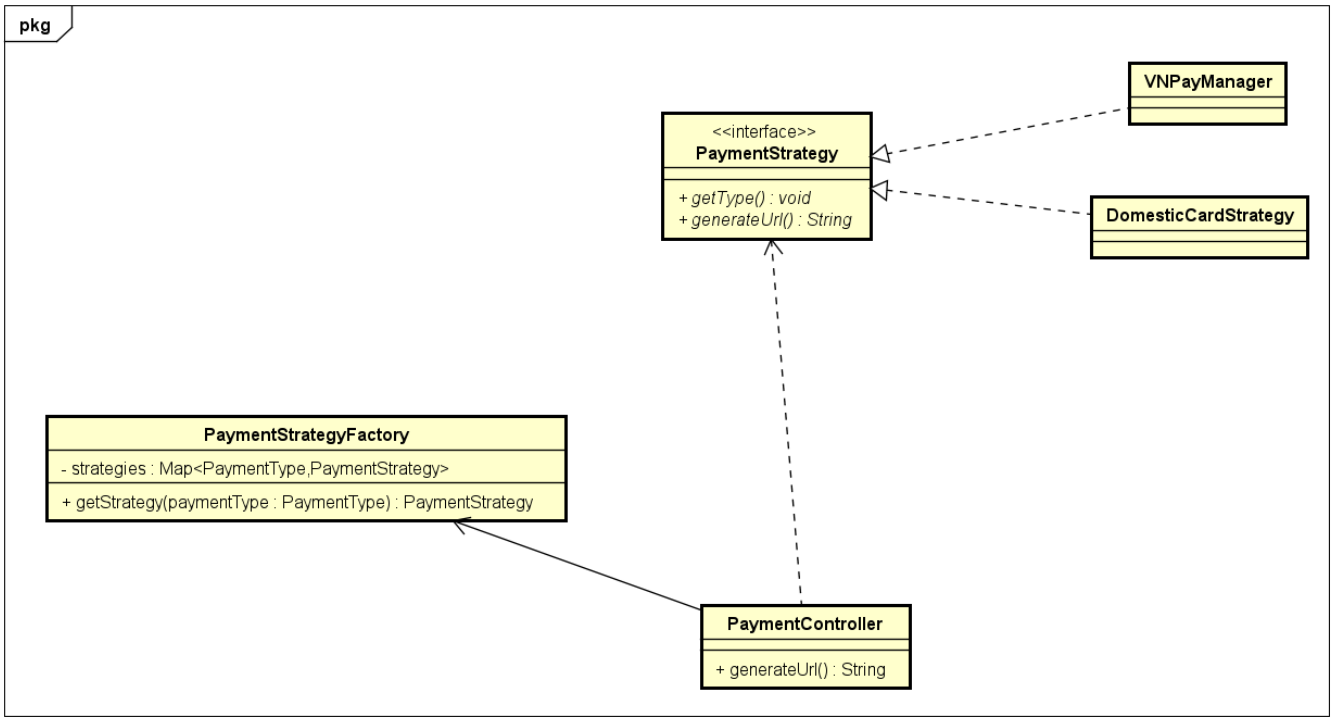
+ Type: Domestic Debit Card
+ Issuing Bank, ví dụ VietinBank
+ Card number 16 kí tự là chữ số
+ Valid-from date, ví dụ: 12/33
+ Cardholder’s name, ví dụ: DO MINH HIEU

Với thẻ nội địa, chương trình vẫn kết nối với API của Interbank, chỉ thay đổi thông tin của phương thức thanh toán.

Vấn đề: control coupling và đồng thời vi phạm tính Open-Close Principle tiếp tục có thể xảy ra minh họa với đoạn code sau:

```
if(paymentType == PaymentType.VNPAY) {
    String result = vnpayService.generateUrl(data);
    return ResponseUtil.success200Response("Success", result);
} else if(paymentType == PaymentType.DOMESTIC_CARD) {
    // do something
}   You, Moments ago • Uncommitted changes
```

Để giải quyết vấn đề trên nhóm đã áp dụng strategy pattern.



Đối với mỗi yêu cầu thanh toán

```
@PostMapping("/pay")
public ResponseEntity<AIMSResponse<Object>> generateUrl(@RequestBody Map<String, Object> data,
                                                         @RequestParam(defaultValue = "VNPAY") PaymentType paymentType) throws IOException {
    PaymentStrategy strategy = paymentFactory.getStrategy(paymentType);
    String result = strategy.generateUrl(data);
     return ResponseUtil.success200Response( message: "Success", result);
}
```

Trong đó triển khai các strategy như sau:

```java
public interface PaymentStrategy {  10 usages  2 implementations
    String generateUrl(Map<String, Object> data) throws IOException;  2

    PaymentType getType();  1 usage  2 implementations        Tan-Tien-Le, 1
}
```

```java
@Service  2 usages
public class PaymentStrategyFactory {
    private final Map<PaymentType, PaymentStrategy> strategies;  2 usages

    public PaymentStrategyFactory(List<PaymentStrategy> paymentStrategies) {
        strategies = paymentStrategies.stream()
                .collect(Collectors.toMap(
                        PaymentStrategy::getType,
                        PaymentStrategy strategy -> strategy
                ));
    }

    public PaymentStrategy getStrategy(PaymentType type) {  1 usage
        PaymentStrategy strategy = strategies.get(type);
        if (strategy == null) {
            throw new PaymentException("No strategy found for payment type: " + type);
        }
        return strategy;   Tan-Tien-Le, 1/1/2025 2:10 PM • feat: strategy for domestic card
    }
}
```

```java
public class DomesticCardStrategy implements PaymentStrategy {  no usages

    @Override  2 usages
    public String generateUrl(Map<String, Object> data) throws IOException {
        return "";
    }

    @Override  1 usage
    public PaymentType getType() { return PaymentType.DOMESTIC_CARD; }

}
```

```java
@Service
public class VNPayManager implements PaymentStrategy {   Tan-Tien-Le, 1/1/2025 2:10 PM • feat: str

    public PaymentTransaction savePaymentTransaction(Map<String, String> response) {  1 usage
        return new PayResponse(response).savePaymentTransaction();
    }

    @Override  2 usages
    public String generateUrl(Map<String, Object> data) throws IOException{
        int amount = (int) data.get("amount");
        String orderId = (String) data.get("orderId");
        PayRequest payRequest = new PayRequest(amount, orderId);
        return payRequest.generateURL();
    }

    @Override  1 usage
    public PaymentType getType() { return PaymentType.VNPAY; }
```

VNPayManager và DomesticCard sẽ implements phương thức generateUrl() , chi tiết triển khai và các thuật toán mã hóa hoặc xác thực cụ thể sẽ được implement trả về đường dẫn trực tiếp đến cổng VNPAY hoặc trang thanh toán của thẻ nội địa.

4. Các design pattern bổ sung

4.1. Singleton

Nhóm đã triển khai Singleton bên frontend để đảm bảo mỗi lần vào trang web người dùng chỉ làm việc với 1 cart duy nhất. Đồng thời tránh các vấn đề về common coupling khi delivery và order đều có quan hệ với cart và sử dụng các trường thông tin trong cart để sử dụng

```jsx
<CartContext.Provider
  value={{
    item,
    setItem,
    cartId,
    totalPrice,
    setTotalPrice,
    shippingPrice,
    setShippingPrice,
  }}
>
  {children}
</CartContext.Provider>
```

4.2. Builder Pattern

Ngoài ra nhóm cũng sử dụng builder pattern trong project để cung cấp một cách linh hoạt để tạo đối tượng mà không cần phải sử dụng nhiều constructor (tránh "constructor telescoping").

```java
@Override  1 usage
public Pagination<Object> findAllProduct(int page, int limit) {
    var pageAble = PageRequest.of(page, limit);
    Page<Product> products = productRepository.findAll(pageAble);
    return Pagination.builder().
            data(products.getContent()).
            totalPage(products.getTotalPages()).totalItems(products.getTotalElements()).build();
}
```