**Bài thực hành 05 – Tuần 13**

* Link đã nộp bài lên Git/GitHub:

<https://github.com/SuPerEle14/TranTheLam-20183937-710809>

* Link Source code:

<https://github.com/SuPerEle14/TranTheLam-20183937-710809.git>

**Báo Cáo 05: Các khái niệm và nguyên lý thiết kế phầm mềm**

## MỤC ĐÍCH VÀ NỘI DUNG

* Trong bài thực hành này, người học sẽ làm quen với các khái niệm thiết kế (Design Concepts) và các nguyên lý cơ bản trong thiết kế (Design Principles)
* Đối với Design Concepts, người học sẽ được tiếp cận với 2 khái niệm chính là cohesion và coupling.
* Đối với Design Principles, người học sẽ được làm quen với SOLID, đó là 5 chữ cái viết tắt của 5 nguyên lý cơ bản trong thiết kế: Single responsibility principle, Open/Closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle.
* Sau bài thực hành, người học sẽ có thêm những ý tưởng để tạo ra một bản thiết kế tối ưu, dễ bảo trì, mở rộng…

## CHUẨN BỊ

* Người học cần hiểu được các kiến thức lý thuyết về Design Concept và Design Principle đã được học ở trên lớp trước khi bắt tay vào thực hành.

## NỘI dung chi tiết

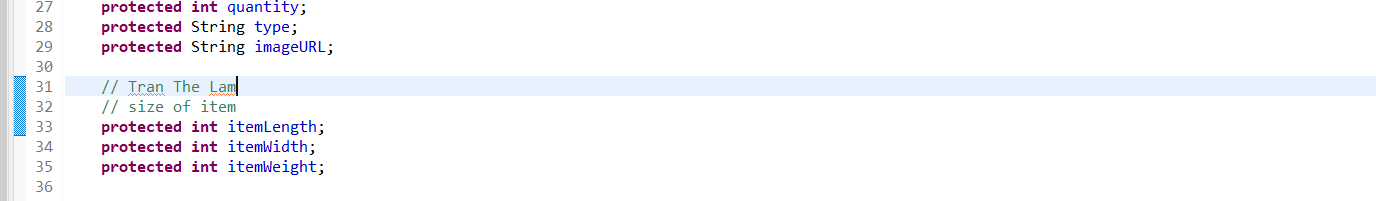
### Một số yêu cầu mở rộng

Dưới đây là danh sách các yêu cầu mở rộng:

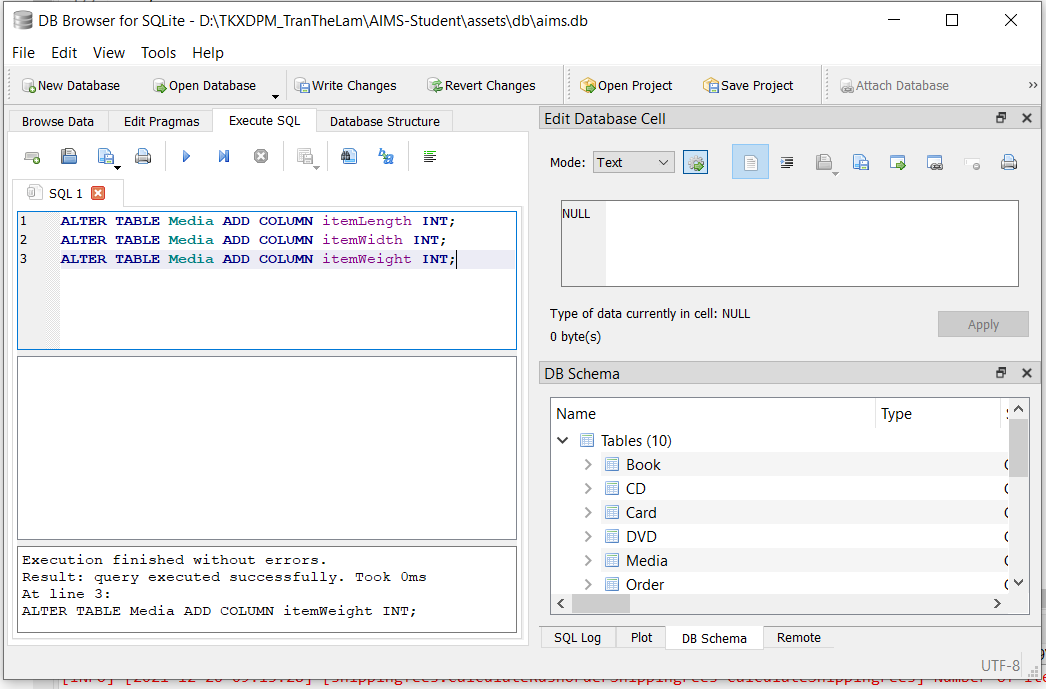
1. **Thay đổi phí ship:**

* Phí vận chuyển hiện nay cũng phụ thuộc vào trọng lượng thực tế, khoảng cách và độ cồng kềnh của sản phẩm (tức là kích thước của sản phẩm: chiều dài, chiều rộng, chiều cao)
* Công thức chuyển đổi được đưa ra như sau:
* Alternative weight (kg) =Length (cm) × Width (cm) × Height (cm) / 6000
* Trọng lượng của sản phẩm bằng trọng lượng thực tế cộng với trọng lượng thay thế

Thêm 3 trường chiều dài(length), chiều rộng(width), chiều cao(height) vào Media entity để sử dụng thể hiện cách tính tiền

. 

Thêm 3 trường dữ liệu vào Database:



Ảnh có chứa bàn

Mô tả được tạo tự động

1. **Thêm một cách tính tiền mới:**

* Domestic debit card:
* Issuing bank, e.g., VietinBank
* Card number: 16 digits
* Valid-from date, e.g., 12/33
* Cardholder’s name, e.g., VU DUY MANH
* Đối với các yêu cầu mở rộng này, giả sử API vẫn giữ nguyên, giả sử thông tin thẻ được thay đổi.

Sử dụng mẫu strategy pattern để thực hiện nhiều phương thức thanh toán

Ảnh có chứa văn bản

Mô tả được tạo tự động

1. **Sử dụng interbank khác**

* Sử dụng một interbank khác với giao thức kết nối và API thay đổi.

### Coupling và Cohesion: các khái niệm cơ bản trong thiết kế

#### Coupling

* Để có một bản thiết kế tốt thì cần phải thiết kế sao cho coupling lỏng để khi có sự thay đổi ở một module thì ảnh hưởng ít nhất có thể đến các module khác. Coupling càng lỏng lẻo, càng tốt. Trong phần này, chúng ta sẽ xem xét về các mức coupling giữa các module trong một project mẫu.

1. **Content coupling**

* May mắn thay, project của chúng tôi hiện tại chưa có loại mức độ coupling này.
* Để minh chứng cho điều này, hãy xem class Order trong package entity.order. Hiện tại, lớp này có thuộc tính deliveryInfo có kiểu HashMap và một getter public: getDeliveryInfo() cho thuộc tính này. Trong tương lai, có thể có một module quản lý để lấy đối tượng deliveryInfo bằng cách gọi getDeliveryInfo() của lớp Order và sau đó thay đổi deliveryInfo bằng cách gọi put() của lớp HashMap. Do đó, giá trị của deliveryInfo sẽ bị thay đổi trong khi object Order không biết gì về việc sửa đổi thuộc tính của nó từ bên ngoài.
* Để giảm mức độ ghép nối, chúng ta có thể:
  + Bỏ qua những accessors/getters dư thừa
  + Chọn chỉ định truy cập phù hợp.
  + Đóng gói dữ liệu bằng một lớp / đối tượng chuyên biệt.
  + Sử dụng design pattern như builder patterns.
* Hãy nhớ nghĩ đến mức độ phụ thuộc giữa các mô-đun khi thiết kế.

1. **Common Coupling**

* Đây là trượng hợp 2 module cùng chia sẻ chung dữ liệu, 2 global structure có thể cùng vào chỉnh sửa, truy cập hoặc có chung những khối mã nguồn thì sẽ vi phạm common coupling.
* Tuy nhiên, object oriented programming không có common data. Tất cả data đều thuộc về lớp. Ví dụ khi dùng C, bạn có data structure khai báo là global, sau đó sẽ có một số phương thức ghi vào structure, các phương thức khác đọc từ đó, trường hợp này sẽ vi phạm common coupling. Còn trong Object Oriented, không có bất cứ data nào mà không thuộc về lớp. Do đó, project hiện tại (JAVA) không vi phạm common coupling.

1. **Control Coupling**

* Một module vi phạm control coupling khi nó truyền tham số điều khiển cho các module khác thông qua việc gọi method. Điều này không tốt vì thành phần được gọi sẽ biết được cấu trúc bên trong thành phần gọi và khi cấu trúc này bị thay đổi thì thành phần được gọi sẽ phải thay đổi theo.
* Trong trường hợp sau đây, điều gì sẽ xảy ra nếu có nhiều phương thức thanh toán.
* Thông thường, chúng ta sẽ sử dụng control structure như if-else và sẽ gặp phải vấn đề về control coupling như sau:
* Cách xử lý:
  + Tách phương thức
  + Áp dụng các design patterns như strategy pattern hoặc factory pattern

1. **Stamp coupling**

* Hai lớp được coi là vi phạm stamp coupling nếu một lớp gửi một collection hoặc một object dưới dạng tham số và chỉ một vài phần data được sử dụng ở lớp thứ hai. Để giảm thiểu level coupling này, chúng ta truyền đủ những tham số cần thiết trong function gọi. Chú ý rằng, stamp coupling là mức có thể chấp nhận được.

1. **Data coupling**

* Nếu các module của bạn ở mức data coupling, thiết kế của bạn là thiết kế tốt. Đây là level mà chúng ta hướng đến.

1. **Uncoupled**

* Nếu các modules là uncoupled, vui lòng tách chương trình của bạn thành các module độc lập.

#### Cohesion

* Nhìn chung, để tăng mức độ (level) cohesion, chúng ta có thể thử đặt mỗi phần vào một module khác phù hợp hơn (tạo module mới nếu cần)

1. **Coincidental cohesion**

* Các sub component đặt trong 1 component vì tính ngẫu nhiên
* Rõ ràng, chúng ta có thể thấy loại cohesion này trong class Config hoặc class Utils trong package utils

1. **Logical cohesion**

* Các thành phần trong 1 module có liên quan đến nhau nhưng về mặt logic chứ không phải chức năng.
* Quan sát lại phần Control Coupling bên trên, nếu chúng ta chia đoạn code thành 2 methods và đưa chúng vào trong cùng một lớp, chúng ta có thể đối mặt với logical cohesion. Do đó, chúng ta cần xem xét đưa chúng vào trong những class hoặc package khác.

1. **Temporal cohesion**

* Các sub component đặt trong một component vì chúng liên quan đến nhau về mặt thời gian chứ không phải về mặt chức năng.
* Thông thường, chúng ta cần loại cohesion này trong các module khởi tạo hoặc clean-up.
* Ví dụ: viết một phương thức khởi tạo tất cả các thành phần của hệ thống 🡺 vi phạm Temporal cohesion: thành phần này đi khởi tạo dữ liệu cho thành phần khác, thay vì vậy ta nên gọi đến thành phần khởi tạo của từng thành phần.

1. **Procedural cohesion**

* Các thành phần đặt trong 1 module vì nó có quan hệ chặt chẽ với nhau theo một thứ tự nào đó chứ không liên hệ với nhau về mặt chức năng.
* Chúng ta có thể thấy loại cohesion này ở trong class PlaceOrderController trong package control. Chúng ta validate các trường dữ liệu từng bước một với các phương thức validation.

Ảnh có chứa văn bản

Mô tả được tạo tự động Ảnh có chứa văn bản

Mô tả được tạo tự động

1. **Communicational cohesion**

* Các thành phần đặt trong cùng một module vì cùng thực hiện trên cùng một dữ liệu.
* Các module theo kiểu cohesion này hoạt động trên cùng một đầu vào hoặc trả về cùng một đầu ra (ví dụ: các thành phần trong InterbankSubsystemController đều nhận các dữ liệu đầu vào giống nhau và dữ liệu đầu ra cùng trả về kiểu PaymentTransaction)

Ảnh có chứa văn bản

Mô tả được tạo tự độngẢnh có chứa văn bản

Mô tả được tạo tự động

1. **Sequential cohesion**

* Trong một module, output của thành phần này là input của thành phần kia.
* Chỉ có một vấn đề nhỏ ở đây. Nếu chúng ta có 1 trình tự, chúng ta có thể chia nó thành nhiều phần khác nhau. Có nghĩa là các class được tạo ra có thể làm nhiều hơn một chức năng hoặc hoàn toàn ngược lại - chỉ một phần của chức năng.

1. **Informational cohesion**

* Các operation có tính độc lập (có input và output riêng nhưng chúng có thể thao tác trên một tập dữ liệu chung là attribute của lớp đó)
* Chúng ta có thể thấy rõ loại cohesion này ở trong các lớp entity như là Media hay Order.

1. **Functional cohesion**

* Mỗi một subcomponent thực hiện một công việc nào đó và hướng đến mục đích chung của component đó.
* Nhìn lại class API sau khi đã được refactor ở bài lab trước, bạn có thể thấy rõ được, đầu ra phương thức setUpConnection() là đầu vào cho phương thức get().
* Ảnh có chứa văn bản

  Mô tả được tạo tự động
* Ảnh có chứa văn bản

  Mô tả được tạo tự động

Ảnh có chứa văn bản

Mô tả được tạo tự động

### Nguyên lý thiết kế SOLID

#### Single Responsibility

* Một lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi. Một lớp quá quá nhiều chức năng sẽ trở nên cồng kềnh, khó đọc, khó bảo trì, nên chỉ có duy nhất một lý do để thay đổi một lớp, không nên có 2 lý do trở lên.
* Như đã nêu trong bài thực hành Class Design, InterbankSubsystemController chịu trách nhiệm cho 2 nhiệm vụ: (1) điều khiển luồng dữ liệu (2) chuyển đổi dữ liệu (chuyển đổi dữ liệu nhận về từ api sang dạng controller yêu cầu). Do đó, lớp này phải được thay đổi khi mà luồng dữ liệu thay đổi (ví dụ các tính năng mới được thêm vào) hoặc cách chuyển đổi dữ liệu thay đổi (ví dụ như thay đổi dữ liệu định dạng bắt buộc), đây là một chỉ báo của một bản thiết kế chưa tốt.
* Nguyên tắc này nghe có vẻ đơn giản nhưng việc phát hiện ra nó và thực hiện nó là rất khó và phức tạp.

#### Open/Closed

* Theo nguyên lý này, mỗi khi chúng ta thêm mới chức năng chúng ta nên viết class mới extend từ class đã có chứ không nên chỉnh sửa trực tiếp nội dung trên class đã viết.
* Sau khi đã hiểu về nguyên lý này, bạn dễ dàng nhận thấy chúng ta đã tuân thủ nguyên tắc này khi thiết kế subsystem cho interbank. Sau này mở rộng, chúng ta có thể sử dụng interbank khác với giao thức kết nối và API thay đổi thì chúng ta chỉ cần viết một subsystem khác implement các phương thức payOrder, refund … Điều này đáp ứng được tính open/closed, tức là thay đổi yêu cầu mà không phải sửa lại thiết kế cũ.

Description: Diagram

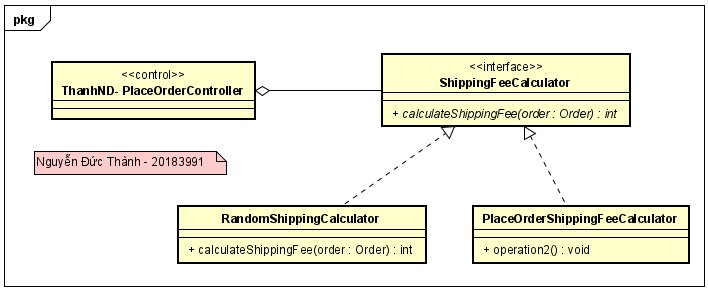
Description automatically generated

* Ngoài ra, đối với thiết kế cũ, bạn có thể thấy lớp PlaceOrderController có một phương thức là calculateShippingFee như sau:

Description: Text

Description automatically generated

* Sau khi implement code, chúng ta có thể dễ dàng nhận thấy, sau này, nếu ta muốn tính phí ship theo kiểu khác, hoặc cần lưu nhiều kiểu tính phí ship khác nhau, thì chúng ta sẽ phải sửa đoạn code trên bằng một đoạn code với cách tính hoàn toàn khác. Điều này vi phạm nguyên tắc open/closed
* Giải pháp: ta sẽ tạo ra một interface ShippingFeeCalculator với phương thức trừu tượng là calculateShippingFee. Khi chúng ta muốn thêm một hay nhiều cách tính phí ship thì chúng ta chỉ cần viết thêm một lớp mới implement interface trên. Và tại PlaceOrderController chúng ta khởi tạo một đối tượng là interface ShippingFeeCalculator với instance là loại chiến lược tính phí ship mà chúng ta muốn. Vậy là đã đáp ứng được nguyên lý open/closed.



Ảnh có chứa văn bản

Mô tả được tạo tự động

Ảnh có chứa văn bản

Mô tả được tạo tự động

Ảnh có chứa văn bản

Mô tả được tạo tự động

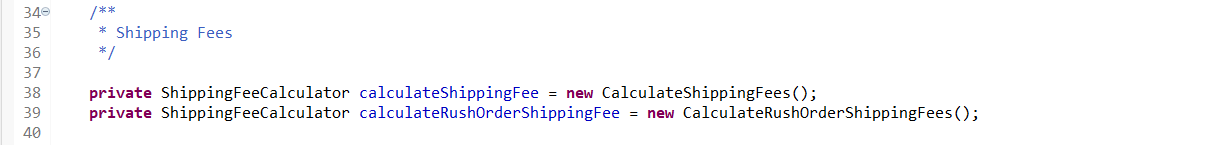
Ảnh có chứa văn bản

Mô tả được tạo tự động

Ảnh có chứa văn bản

Mô tả được tạo tự động

Thực hiện implements vào PlaceOderController

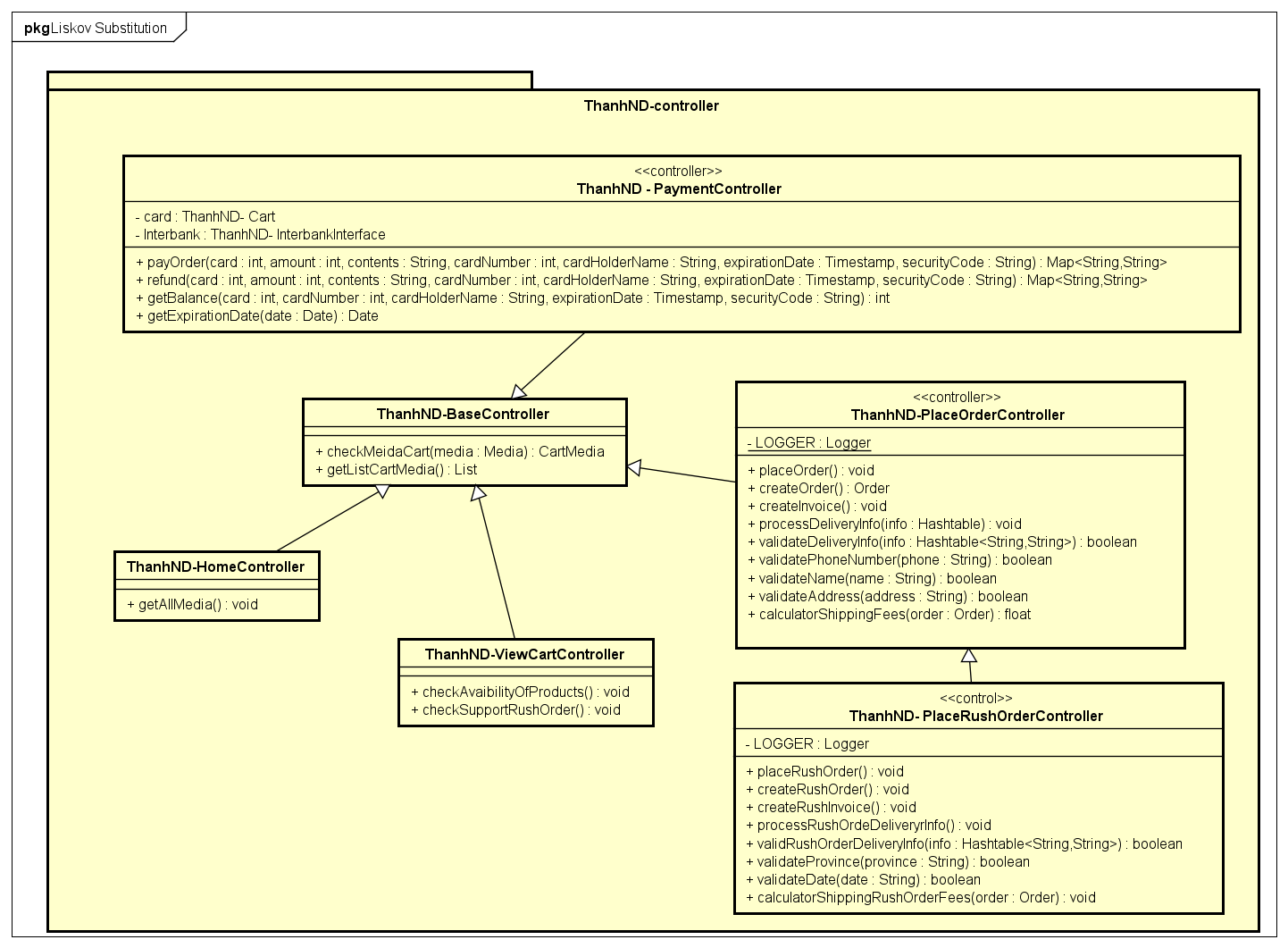


Ảnh có chứa văn bản

Mô tả được tạo tự động

#### Liskov Substitution

* Nguyên tắc này nói rằng, các đối tượng của class con có thể thay thế cho lớp cha ở mọi tính huống mà không gây ra lỗi, hoặc nếu không, chúng ta có sự trừu tượng sai.
* Có thể thấy hệ thống phân cấp kế thừa ở class BaseController đã tuân theo nguyên lý này.



* Bây giờ hãy nhìn vào mã nguồn và xem xét hệ thống phân cấp cây kế thừa của Media có vi phạm nguyên tắc này không?

Description: Diagram

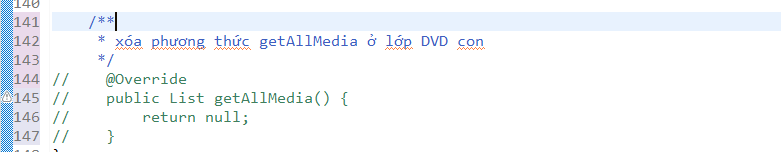
Description automatically generated

* Ta có thể thấy phương thức Media.getAllMedia() được kì vọng trả về một List, tất cả các class con override lại phương thức này nhưng lại trả về null

Class Media:



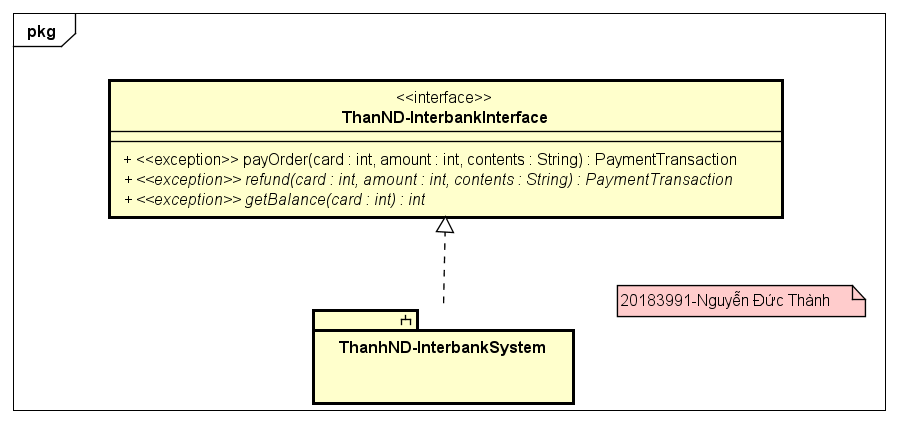
Class DVD(tương tự với class Book và CD)



* Giải pháp: xoá đoạn code Override đi. Chương trình sau đó vẫn chạy bình thường vì phương thức getAllMedia() của lớp cha Media không phải phương thức abstract, không bắt buộc phải override.

#### Interface Segregation

* Nguyên lý này nói rằng, thay vì một sử dụng một interface quá lớn, quá nhiều phương thức thì chúng ta sẽ tách nhỏ ra thành các interface con với mục đích cụ thể. Vì khi để một interface quá to, các lớp implement sẽ phải implement các phương thức mà bản thân nó không cần dùng đến.



* Thiết kế hiện tại về cơ bản đã đáp ứng được nguyên tắc này, ví dụ với InterbankInterface, cả 2 phương thức payOrder và refund đều được lớp InterfaceSystemController implement.
* Tuy nhiên, cũng cần phải cân nhắc, bởi trong tương lai, có thể có một số hệ thống interbank khác không hoàn tiền cho khách hàng mà chỉ thanh toán, lúc này phương thức refund của InterbankInterface trở nên dư thừa đối với interbanksubsystem đó 🡺 vi phạm Interface Segregation.

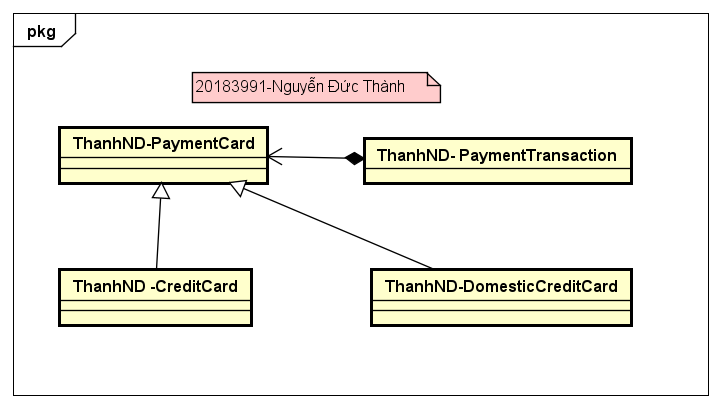
#### Dependency Inversion

* Có thể hiểu nguyên lý này như sau: những thành phần trong một chương trình chỉ nên phụ thuộc vào những cái trừu tượng. Những thành phần trừu tượng không nên phụ thuộc vào một thành phần mang tính cụ thể mà nên ngược lại.

Description: Diagram

Description automatically generated

* Hiện tại, PaymentTransaction đang phụ thuộc chặt chẽ vào lớp CreditCard, sau này giả sử không sử dụng CreditCard để thanh toán mà sử dụng một loại phương thức thanh toán khác, ví dụ như domestic debit card… như vậy thiết kế hiện tại đã vi phạm nguyên lý D trong SOLID.
* Giải pháp: tạo một lớp abstract là PaymentCard và lớp PaymentTransaction chỉ quan tâm đến lớp PaymentMethod này.



Ảnh có chứa văn bản

Mô tả được tạo tự động

Ảnh có chứa văn bản

Mô tả được tạo tự động

Lớp CreditCard

Ảnh có chứa văn bản

Mô tả được tạo tự động

Lớp DomesticCreditCard

Ảnh có chứa văn bản

Mô tả được tạo tự độngẢnh có chứa văn bản

Mô tả được tạo tự động

Lớp PaymentTransaction

## BÀI TẬP

Review Design sau khi đã thêm UC “Place Rush Order”

### Coupling và Cohesion

**- Review lại các mức coupling và cohesion và xác định các mức độ này trong modules của thiết kế cho“UC Place Rush Order”. Nếu thiết kế chưa tốt, hãy đề xuất giải pháp cải thiện.**

**- Viết báo cáo cho các vấn đề trên và sửa lại thiết kế và source code tương ứng với đề xuất.**

Sau khi hoàn thành xong, nộp lại kết quả vào thư mục “GoodDesign/DesignConcepts.”

Dưới đây là một báo cáo mẫu:

1. Coupling
   1. Content coupling

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| Order | Hiện tại thuộc tính deliveryInfo đang bị truy cập sửa chữa ngoài phạm vi cho phép | Dự định Builder Pattern để giải quyết |

* 1. Common coupling

Vì JAVA là ngôn ngữ thuần OOP nên không có đối tượng nào chia sẻ chung dữ liệu

🡪 đảm bảo được nguyên tắc này.

* 1. Control coupling

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| calculateRushOrderFees(Order  order) | Hiện tại chưa check vị trí người đặt hàng đối với Rush Order, mới chỉ xét xem món hàng đó có hỗ trợ đặt hàng nhanh không thôi. | Thêm biến Check địa chỉ + các biến xác định khoảng cách giữa các tỉnh |

* 1. Stamp coupling

Chỉnh sửa Coding Convention

1. Cohesion
   1. Coincidental cohesion

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| PlaceOrderController | Phương thức tính tiền được đặt ngay trong lớp này | Chuyển thành một gói tính tiền riêng |

* 1. Logic cohension

Chưa tìm thấy vấn đề nào

* 1. Temporal cohension

Các class được khởi tạo theo trình tự xuất hiện của nó (Home 🡪 Cart 🡪 Order 🡪 Invoice 🡪 Success)

* 1. Procedural cohension

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| PlaceOrderController | Các hàm Validate không cùng thực hiện một chức năng mà chỉ có quan hệ chặt chẽ với nhau là đều dung để kiểm tra dữ liệu người dung nhập vào. | Thỏa mãn điều kiện |
| PlaceRushOrderController | Các hàm Validate không cùng thực hiện một chức năng mà chỉ có quan hệ chặt chẽ với nhau là đều dùng để kiểm tra dữ liệu người dùng nhập vào. | Thỏa mãn điều kiện |

* 1. Commucation cohension

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| PaymentController | Các phương thức refund và payOrder đều trả về cùng một PaymentTransaction | Thỏa mãn điều kiện |
| InterbankSubsystemController | Các thành phần trong subsystem đều nhận các dữ liệu đầu vào giống nhau và đều trả về kiểu PaymentTransaction | Thỏa mãn điều kiện |

* 1. Sequential cohension

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| CalculateShippingFees | Tính giá đơn hàng  1. Kiểm tra xem đơn hàng có quá 1.000.000 vnd không  2. tính giá tiền vẫn chuyển | Thỏa mãn điều kiên |
| CalculatePlaceRushOrderShippingFees | Tính giá trị đơn giao hàng nhanh  1. Đếm số Mặt hàng chọn phương thức giao hàng nhanh  2. Tính phí theo số lượng mặt hàng chọn giao hàng nhanh | Thỏa mãn điều kiện |
| CalculateNewShipping  Fees | Tính giá trị đơn hàng theo trọng lượng  1. Lấy kích thước đơn hàng  2. Lấy Trọng lượng thực tế  3. Tính trọng lượng thay thế /6000  4. Tính phí giao hàng mới | Thỏa mãn điều kiện |
| Các lớp handler | Truyền nhau giữa các đối tượng | Thỏa mãn điều kiện |

* 1. Information cohension

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| Media, CD, DVD, các lớp handler, PlaceOrderController, PlaceRushOrderController | Các phương thức đều độc lập nhưng chia sẻ chung attribute của lớp | Thỏa mãn điều kiện |

* 1. Function cohension

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
|  |  |  |
| CalculatePlaceRushOrderShippingFeesr | Tính giá trị đơn giao hàng nhanh  1. Đếm số Mặt hàng chọn phương thức giao hàng nhanh  2. Tính phí theo số lượng mặt hàng chọn giao hàng nhanh | Thỏa mãn điều kiện |

### Nguyên lý thiết kế SOLID

**Trong phần này, bận cần thực hiện:**

**- Kiểm tra xem design của bạn sau khi đã thêm UC “Place Rush Order” xem có đáp ứng nguyên lý SOLID hay không trước và sau khi thêm các yêu cầu mở rộng. Nếu thiết kế chưa tốt, hãy đề xuất giải pháp giúp cho thiết kế tốt hơn để đáp ứng được với 5 nguyên lý SOLID.**

**- Viết báo cáo về các vấn đề trên và chỉnh sửa code để implement đề xuất của bạn. Chú ý rằng bạn không cần phải implement các yêu cầu mở rộng. Bạn cần cân nhắc việc có cần thiết kế để đảm bảo có các yêu cầu mở rộng thì không vi phạm các nguyên lý trong SOLID.**

Sau khi hoàn thành xong, nộp lại kết quả vào thư mục “GoodDesign/DesignPrinciples.”

1. Single Responsibility Principle

|  |  |  |  |
| --- | --- | --- | --- |
| # | Related modules | Description | Improvement |
|  | PlaceOrderController | Có nhiều chức năng trong cùng một lớp | Tách phần tích phí ra một package riêng |

1. Open/Closed Principle

|  |  |  |  |
| --- | --- | --- | --- |
| # | Related modules | Description | Improvement |
|  | PlaceOrderController, PlaceRushOrderController | Cứng nhắc trong việc tính phí Giao hàng 🡪 Khó bảo trì hệ thống | Tạo ra các interface và các lớp tính phí implement interface này) |

Ảnh có chứa văn bản

Mô tả được tạo tự động

1. Liskov Substitution

|  |  |  |  |
| --- | --- | --- | --- |
| # | Related modules | Description | Improvement |
|  | Media | Phương thức getAllMedia ở các lớp con trả về null thay vì trả về một list như ở lớp cha |  |

1. Interface segregation

|  |  |  |  |
| --- | --- | --- | --- |
| # | Related modules | Description | Improvement |
|  | PlaceOrderScreenHandler, PlaceRushOrderScreenHandler | Phương thức notifyError ở các lớp con trả về null | Xóa bỏ các lớp con override lại phương thức notifyError |

5. Dependency Inversion

|  |  |  |  |
| --- | --- | --- | --- |
| # | Related modules | Description | Improvement |
|  | PaymentTransaction | PaymentTransaction phụ thuộc vào CreditCard | Tạo một lớp abstract là PaymentCard và lớp PaymentTransaction chỉ quan tâm đến lớp này |

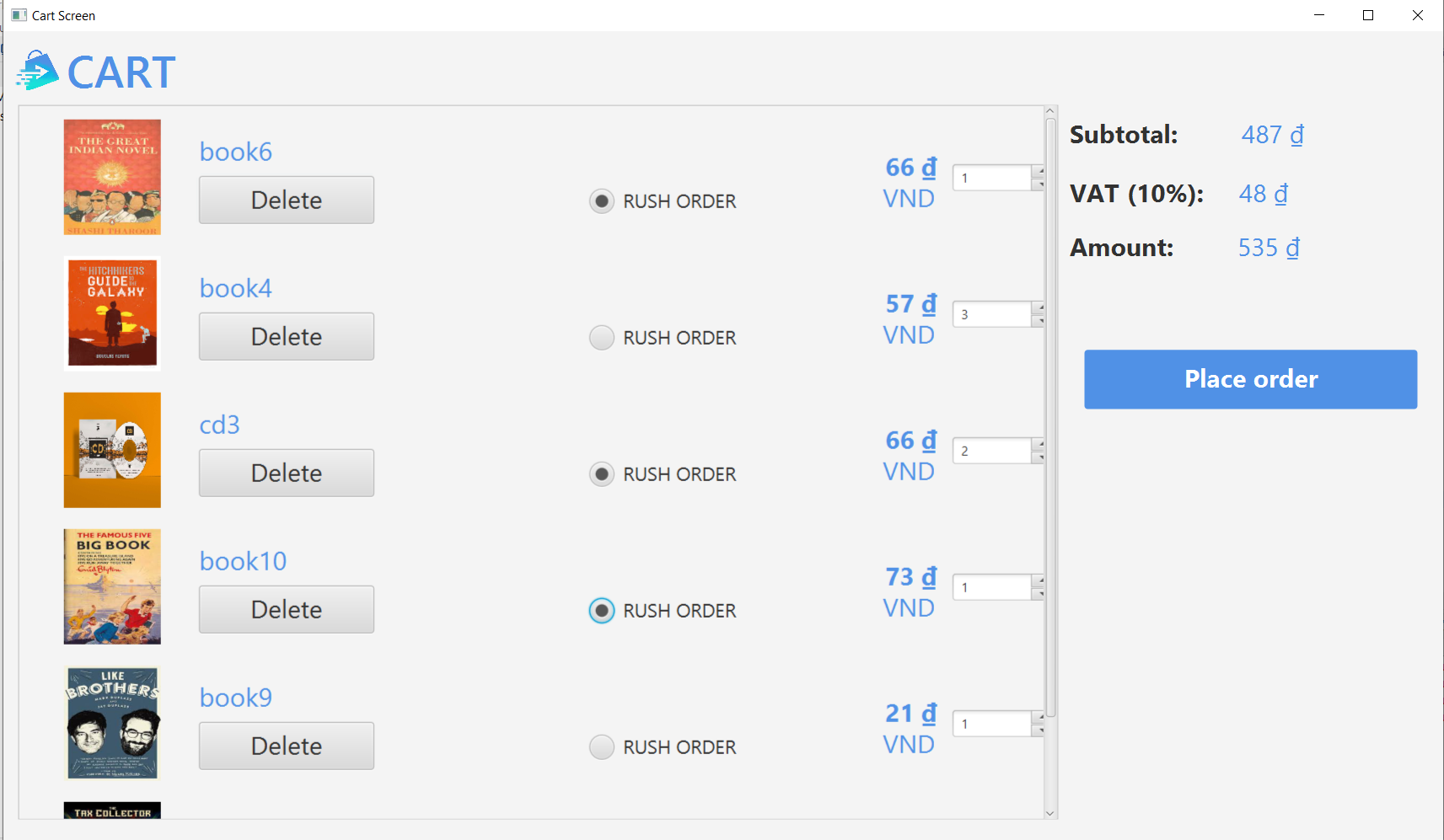
Ảnh có chứa văn bản

Mô tả được tạo tự động

**Chương trình chạy Khá tốt sau khi áp dụng các nguyên lý cohension, coupling, với các design pattern**

Một số hình ảnh về sản phẩm

Ảnh có chứa văn bản, ảnh chụp màn hình, trắng, vài

Mô tả được tạo tự động  Ảnh có chứa văn bản

Mô tả được tạo tự động 