Assignment 12

CÁC KHÁI NIỆM VÀ NGUYÊN LÝ THIẾT KẾ PHẦN MỀM

1. 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…

2. 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.
- Lưu ý các hướng dẫn dưới đây chỉ là 1 số gợi ý chứ không phải là tất cả những gì cần tối ưu cho bản thiết kế và mã nguồn.
- Bản thiết kế và mã nguồn được xem xét trong bài thực hành này có thể clone trên https://github.com/leminhnguyen/AIMS-Student

3. NỘI DUNG CHI TIẾT

1. Bài tập cá nhân

Câu 1: Một lập trình viên thiết kế một class class chứa các phương thức xử lý liên quan đến CSDL như sau:

```
public class DBHelper {
    public Connection openConnection() {};
    public void saveUser(User user) {};
    public List<Product> getProducts() {};
    public void closeConnection() {};
}
```

Thiết kế này có vi phạm nguyên lý nào của SOLID? Giải thích và đưa ra giải pháp.

Trả lời

Không tuân theo nguyên lý solid vì theo nguyên lý single responsibility mỗi class chỉ có 1 chức năng, ở đây class DBHelper vừa có chức năng connect với DB lại có chức năng lấy các sản phẩm getProducts.

Câu 2: Một lập trình viên thiết kế logic xử lý tính phí vận chuyển của một đơn hàng được đặt luôn bên trong class *Order* như sau:

```
public class Order {

    public long calculateShipping(ShippingMethod shippingMethod) {
        if (shippingMethod == GROUND) {
            // Calculate for ground shipping
        } else if (shippingMethod == AIR) {
            // Calculate for air shipping
        } else {
            // Default
        }
    }
}
```

Giả sử hệ thống cần bổ sung thêm một phương thức vận chuyển mới, điều này khiến cho lập trình viên phải bổ sung một case nữa trong phương thức calculateShipping(). Thiết kế này có vi phạm nguyên lý nào của SOLID? Giải thích và đưa ra giải pháp.

Trả lời

Thiết kế này có vi phạm nguyên lý SOLID. Theo nguyên lý Open-close 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.

Câu 3: Một lập trình viên thiết kế một giao diện chứa các phương thức phục vụ cho việc chuyển đổi các kiểu dữ liệu như sau:

```
public interface Conversion {
    public void intToDouble();
    public void intToChar();
    public void charToString();
}
```

Thiết kế này có vi phạm nguyên lý nào của SOLID? Giải thích và đưa ra giải pháp.

Trả lời
Vi phạm nguyên lý Interface Segration. Vì khi một class implement interface Conversion thì phải cài đặt các phương thức như intToDouble(), intToChar(), charToString() mà có thể khi implement các phương thức còn lại không được sử dụng tới.

Câu 4: Một dự án gồm 2 module cấp thấp *BackendDeveloper* và *FrontendDeveloper* và 1 module cấp cao *Project* sử dụng 2 module trên:

```
public class BackendDeveloper {
    public void codeJava() {};
}

public class FrontendDeveloper {
    public void codeJS() {};
}

public class Project {
    private BackendDeveloper backendDeveloper = new BackendDeveloper();
    private FrontendDeveloper frontendDeveloper = new
FrontendDeveloper();
    public void build() {
        backendDeveloper.codeJava();
        frontendDeveloper.codeJS();
    }
}
```

Thiết kế này có vi phạm nguyên lý nào của SOLID? Giải thích và đưa ra giải pháp.

Trả lời

Vi phạm nguyên lý Dependency Inversion. Vì các module cấp thấp giao tiếp với module cấp cao thông qua abstract chứ không nên giao tiếp một cách trực tiếp. Vì khi backend hay frontend thay đổi thì nó sẽ ảnh hưởng đến Project.

2. Một số yêu cầu mở rộng trong project AIMS

Dưới đây là danh sách các yêu cầu mở rộng:

a) 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.

b) 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.

c) Sử dụng interbank khác

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

3. Nguyên lý thiết kế SOLID

3.3.1. Single Responsibility

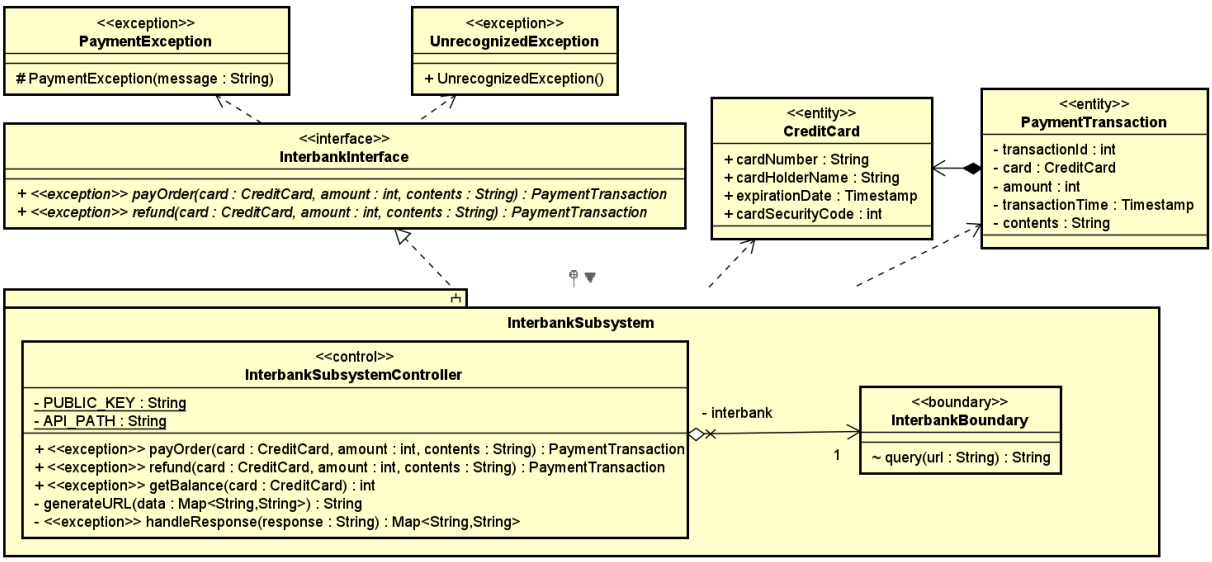
- Một lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi. Một lớp quá quá nhiều chức năng sẽ trở nên cồng kềnh, khó đọc, khó bảo trì, nên chỉ có duy nhất một lý do để thay đổi một lớp, không nên có 2 lý do trở lên.
- Như đã nêu trong bài thực hành Class Design, InterbankSubsystemController 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.

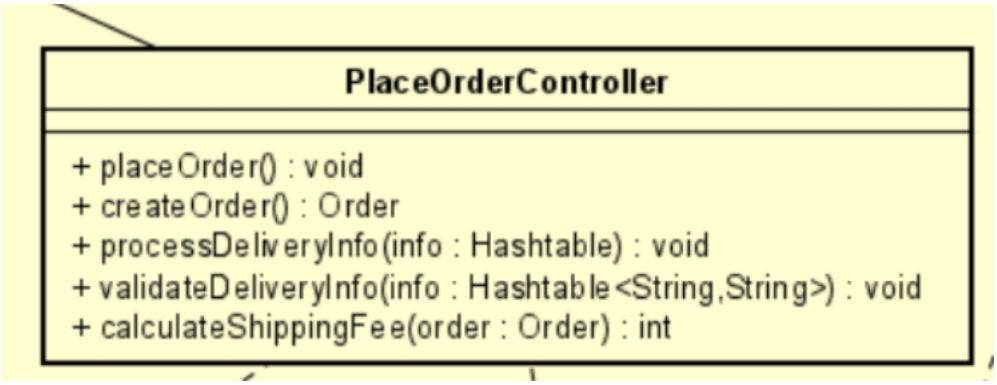
- Nguyên tắc này nghe có vẻ đơn giản nhưng việc phát hiện ra nó và thực hiện nó là rất khó và phức tạp.

3.3.2. 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.



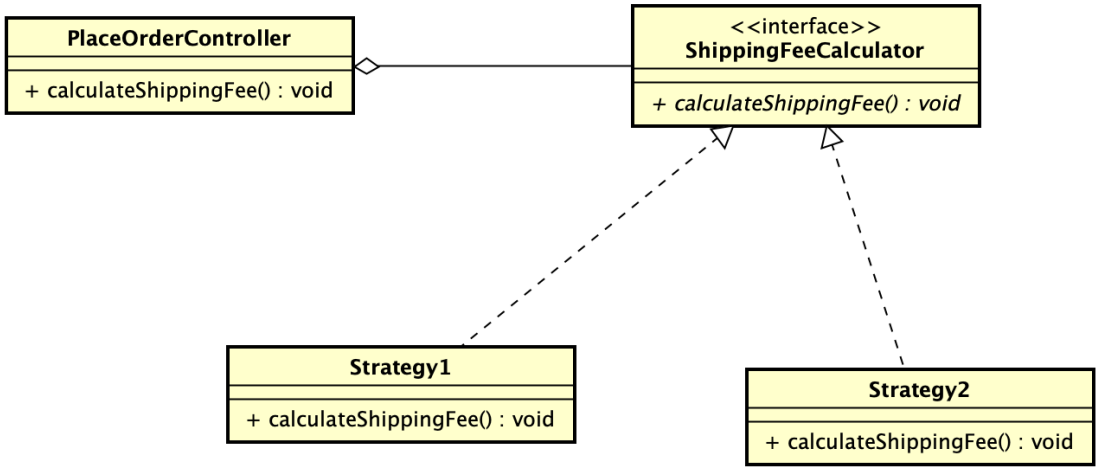
- Ngoài ra, đối với thiết kế cũ, bạn có thể thấy lớp PlaceOrderController có một phương thức là calculateShippingFee như sau:



- 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

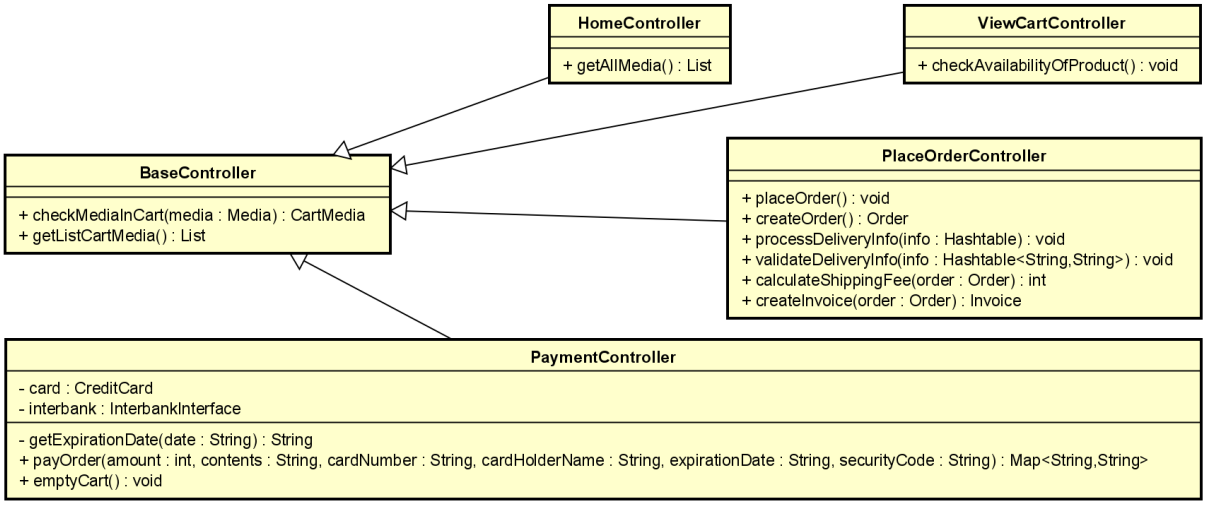
```
public int calculateShippingFee(Order order){
    Random rand = new Random();
    int fees = (int)( ( (rand.nextFloat()*10)/100 ) * order.getAmount() );
    LOGGER.info("Order Amount: " + order.getAmount() + " -- Shipping Fees: " + fees);
    return fees;
}
```

- 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.

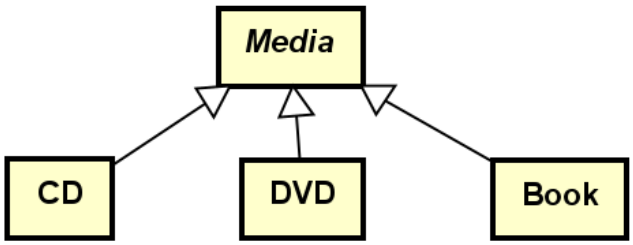


3.3.3. Liskov Substitution

- 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?



- 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:

```
public List getAllMedia() throws SQLException{
    Statement stm = AIMSDB.getConnection().createStatement();
    ResultSet res = stm.executeQuery("select * from Media");
    ArrayList medium = new ArrayList<>();
    while (res.next()) {
        Media media = new Media()
            .setId(res.getInt("id"))
            .setTitle(res.getString("title"))
            .setQuantity(res.getInt("quantity"))
            .setCategory(res.getString("category"))
            .setMediaURL(res.getString("imageUrl"))
            .setPrice(res.getInt("price"))
            .setType(res.getString("type"));
        medium.add(media);
    }
    return medium;
}
```

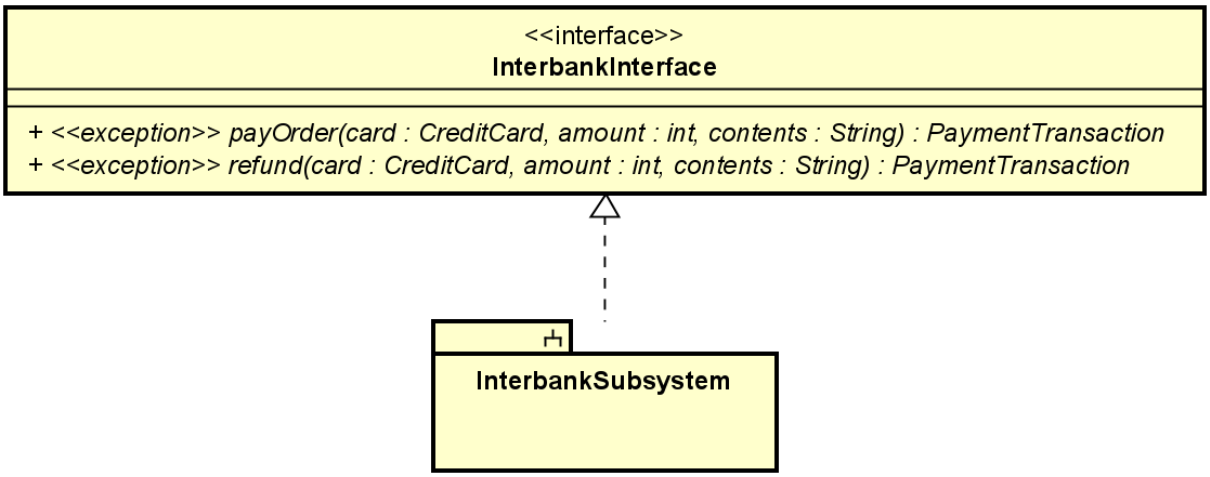
Class DVD:

```
@Override
public List getAllMedia() {
    return null;
}
```

- 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.

3.3.4. *Interface Segregation*

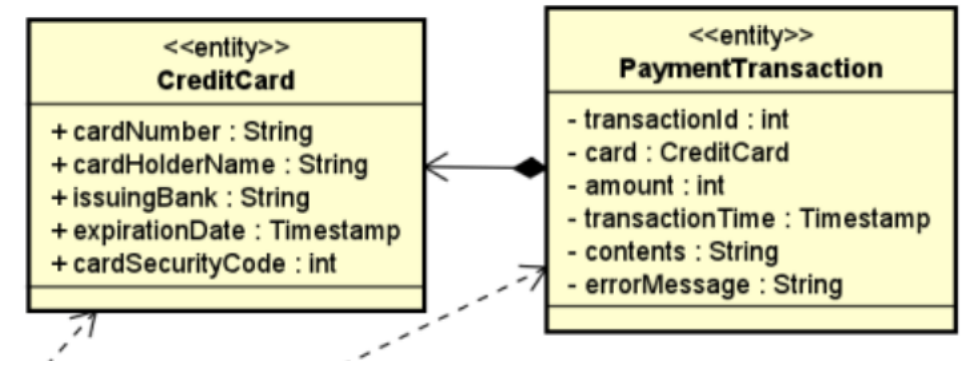
- Nguyên lý này nói rằng, thay vì một sử dụng một interface quá lớn, quá nhiều phương thức thì chúng ta sẽ tách nhỏ ra thành các interface con với mục đích cụ thể. Vì khi để một interface quá to, các lớp implement sẽ phải implement các phương thức mà bản thân nó không cần dùng đến.



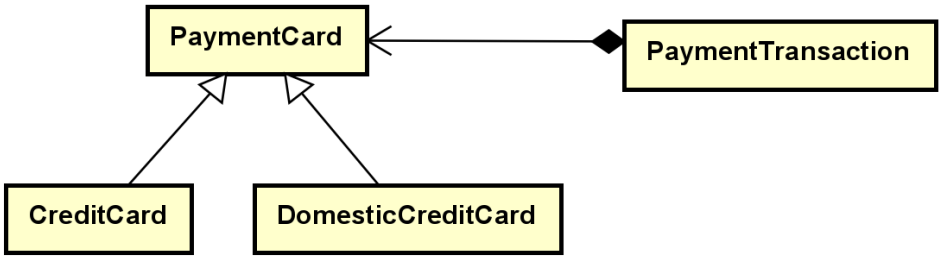
- Thiết kế hiện tại về cơ bản đã đáp ứng được nguyên tắc này, ví dụ với InterbankInterface, cả 2 phương thức payOrder và refund đều được lớp InterfaceSystemController implement.
- Tuy nhiên, cũng cần phải cân nhắc, bởi trong tương lai, có thể có một số hệ thống interbank khác không hoàn tiền cho khách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

3.3.5. 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.



- 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.



- Dễ thấy, giải pháp trên chính là một ví dụ cho cách áp dụng Strategy Pattern vào bản thiết kế.

4. BÀI TẬP CÁ NHÂN

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."

Dưới đây là một báo cáo mẫu:

1. Single Responsibility Principle

#	Related modules	Description	Improvement
1.1.			

2. Open/Closed Principle

#	Related modules	Description	Improvement
2.1.			

...

5. BÀI TẬP NHÓM

Trong phần này bạn cần:

- Kiểm tra xem design 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 cho các vấn đề trên (theo mẫu ở 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."

- Đây là một trong các nội dung quan trọng trong đánh giá kết quả bài tập lớn.

HẾT