ĐẠI HỌC BÁCH KHOA HÀ NỘI

Trường Công nghệ thông tin và Truyền thông

Báo cáo Mẫu thiết kế phần mềm

Nhóm 10

Danh sách thành viên

STT	MSSV	Họ tên
1	20194185	Trịnh Đức Tiệp
2	20194139	Trần Văn Phúc
3	20194018	Chu Thành Đô
4	20194170	Nguyễn Đức Thắng

Hà Nội, 12 tháng 7, năm 2023

Mục lục

1 Tổng quan

Trong báo cáo này, nhóm sẽ tiến hành phân tích kỹ source code của dự án AIMS: An Internet Media Store, để tìm hiểu xem liệu thiết kế hiện tại có bị vi phạm nguyên lý nào trong SOLID hay không, có bị trùng lặp mã nguồn (duplicate) không; hay khi xem xét ở các mức coupling và cohesion, code base có đảm bảo đạt được các mức độ tốt hay không, hay còn những vấn đề nào cần được chỉ ra. Từ đó, nhóm sẽ đề xuất các giải pháp để giúp giải quyết vấn đề, dựa trên các design patterns đã học (Singleton, template method, factory method,...), để vừa gia tăng chất lượng source code, đồng thời tăng cường khả năng tương thích và mở rộng cho các chức năng sau này trong tương lai.

1.1 Tóm tắt kiến thức cơ bản

a) Coupling

- **Content coupling**: Đây là vi phạm coupling ở mức cao nhất được định nghĩa là khi mà một component có thể truy cập trực tiếp vào hoạt động bên trong của một component khác ví dụ như trực tiếp truy xuất data, thay đổi data của component..

- **Common coupling**: Common coupling xảy ra khi mà 2 hay nhiều module cùng đọc và thay đổi 1 dữ liệu dùng chung. Đây cũng là một thiết kế tệ bởi vì nó gây ra sự không rõ ràng về vai trò của dữ liệu, code sẽ khó đọc, khó có thể xác định những thành phần code nào liên quan và ảnh hưởng đến dữ liệu dùng chung. Do đó giảm khả năng bảo trì hay tái sử dụng component.

- **Control coupling**: Xảy ra khi mà tham số truyền vào cho module sẽ quyết định luồng xử lý của module theo những cách khác nhau. Điều này cũng rất thường gặp khi chúng ta truyền tham số vào hàm và tham số đó có thể chia thành nhiều nhóm dữ liệu. Do đó bên trong hàm chúng ta sẽ dùng nhiều if else để check tham số truyền vào. Và với mỗi loại tham số truyền vào thì function đó thực ra chỉ chạy 1 phần code trong 1 block if else thoả mãn điều kiện, và đương nhiên rằng những phần code còn lại là không được chạy.

- **Stamp coupling:** Đến đây thì vi phạm coupling đã không còn ở mức high nữa mà đã có thể chấp nhận được. Stamp coupling xảy ra khi tham số truyền vào cho module là thừa, việc xử lý có thể chỉ cần một vài trường dữ liệu của object nhưng chúng ta truyền nguyên object vào.

- **Data coupling:** Là coupling ở mức thấp nhất khi mà các modules tương tác với nhau chỉ thông qua tham số truyền vào. Điều này là không thể tránh khỏi do vậy thiết kế này là được cho là mục tiêu hướng đến. Data coupling và stamp coupling đều được xác định bởi tham số truyền vào nhưng khác ở chỗ data coupling không

dư thừa dữ liệu tham số truyền vào, tất cả tham số truyền vào đều được sử dụng để xử lý.

b) Cohesion

- **Coincidental cohesion**: Những elements nằm trong component một cách lạc lõng và đơn độc, nó chỉ có điểm chung là nằm chung vị trí với các component khác chứ không liên quan đến mục tiêu thể hiện của component.

- **Logical cohesion**: Được định nghĩa là khi các components liên quan đến nhau một cách logic chứ không phải là liên quan với nhau theo chức năng. Ví dụ như các functions đọc dữ liệu đầu vào từ tape, disk hay network cùng ở chung 1 module nghe có vẻ hợp lý và vì chúng liên quan đến nhau đó là xử lý dữ liệu đầu vào nhưng rõ ràng chức năng của chúng là khác nhau hoàn toàn.

- **Temporal cohesion**: Những elements liên quan đến nhau theo thời gian chứ không theo chức năng và những elements này được thực thi gần như trong cùng một khoảng thời gian.

- **Procedural cohesion**: Những elements liên quan đến nhau chỉ để đảm bảo một thứ tự thực thi cụ thể. Những component được thiết kế elements như vậy vẫn có cohesion ở mức yếu và khó để tái sử dụng.

- **Communicational cohesion**: Là một nhóm các elements của module cùng hoạt động trên cùng một data là dữ liệu I/O của các methods.

- **Sequential cohesion**: Khi output của một element trở thành input của một element khác. Điều này thường xảy ra ở các ngôn ngữ lập trình hướng chức năng (Functional programming languages – Scala, SML, Clojure, Erlang,…). Đây đã được xem như là thiết kế tốt (high cohesion).

- **Functional cohesion**: Đây là mức cao nhất của cohesion khi mà tất cả các elements trong component đều là cần thiết cho mục đích của component.

c) Nguyên lý SOLID

- **Single responsibility principle (SRP)**: Một class nên chỉ có một trách nhiệm duy nhất, và chỉ một lý do để thay đổi.

- **Open/Closed principle (OCP)**: Có thể thoải mái mở rộng một class nhưng không được sửa đổi bên trong class đó.

- **Liskov substitution principle (LSP)**: Bất cứ instance nào của class cha cũng có thể được thay thế bởi instance của class con của nó mà không làm thay đổi tính đúng đắn của chương trình.

- **Interface segregation principle (ISP)**: Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể. Client không nên phụ thuộc vào interface mà nó không sử dụng.

- **Dependency inversion principle (DIP)**: Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction. Abstraction không nên phụ thuộc vào chi tiết, mà ngược lại.

d) Singleton

Single Pattern là một design pattern:

- Đảm bảo rằng một class chỉ có duy nhất một instance.
- Và cung cấp một cách toàn cục để truy cấp tới instance đó.

e) Template method

- **Template Method** xây dựng một bộ khung thuật toán trong một trình tự công việc, để lại việc định nghĩa một vài bước cho các subclass mà không làm thay đổi cấu trúc chung của thuật toán.

f) Factory method

- **Factory method** (hay còn gọi là virtual constructor) là một mẫu thiết kế thuộc nhóm Creational Patterns – những mẫu thiết kế cho việc khởi tạo đối tượng của lớp. Khi chúng ta muốn tạo ra một object của một type nào đấy, nhưng chúng ta không biết rõ mình sẽ phải tạo ra cái gì, mà nó phải dựa vào một số điều kiện business logic đầu vào để tạo ra object tương ứng, thì chúng ta có thể sử dụng Factory Method này.

g) Strategy pattern

- **Strategy pattern** là mẫu thiết kế giúp trừu tượng hóa những hành vi (behavior, method, function) của một đối tượng bằng cách tách những cài đặt vào những lớp khác nhau.

h) Adapter pattern

- **Adapter pattern** là 1 design pattern thuộc nhóm Structural Pattern. Pattern này giúp tạo ra một “cổng chuyển đổi” giữa interface đang sử dụng trong hệ thống với các interface khác ở bên ngoài mà không cần modify lại code đã có.

1.2 Mục tiêu

Mục đích sử dụng của báo cáo mẫu thiết kế phần mềm là cung cấp một tài liệu tổng quan về thiết kế của phần mềm AIMS. Báo cáo này giúp cho các thành viên trong đội phát triển phần mềm hiểu rõ về cấu trúc, cách thức hoạt động và các thành phần của phần mềm. Từ đó giúp mọi thành viên trong đội có thể tham gia vào quá trình phát triển phần mềm một cách nhanh chóng, dễ dàng và chính xác.

Đối tượng người đọc của báo cáo mẫu thiết kế phần mềm là các thành viên trong nhóm phát triển phần mềm, bao gồm các lập trình viên, nhà phát triển, kiến trúc sư phần mềm, người quản trị dự án và những người liên quan khác. Các thành viên này cần có kiến thức chuyên môn về phát triển phần mềm và quan tâm đến chi tiết thiết kế của hệ thống.

Nội dung khái quát của báo cáo mẫu thiết kế phần mềm thường bao gồm:

1. Tổng quan của bài báo cáo. Mục này trình bày những kiến thức cơ bản về Mẫu thiết kế phần mềm, sau đó là những mục tiêu của báo cáo, phạm vi của báo cáo, danh sách các thuật ngữ được sử dụng trong báo cáo và danh mục các tài liệu tham khảo.

2. Đánh giá thiết kế cũ. Trong phần này, nhóm sẽ chỉ ra những nhận định chung về thiết kế cũ của codebase. Sau đó là danh sách các vấn đề cụ thể mà codebase gặp phải như các vấn đề vi phạm coupling, cohesion, các nguyên lý SOLID và các quy tắc của Clean Code cùng một số vấn đề khác có liên quan.

3. Đề xuất cải tiến. Nhóm sẽ đưa ra một loạt các đề xuất để cải tiến cho các vấn đề codebase gặp phải mà nhóm đã phân tích trong phần 2. Mỗi giải pháp có thể sẽ giải quyết được nhiều vấn đề cùng một lúc và đảm bảo mã nguồn sẽ sạch sẽ hơn, thuận lợi hơn cho việc mở rộng và phát triển trong tương lai.

4. Tổng kết. Phần cuối cùng của này là nội dung tổng hợp lại những công việc, kết quả tổng quan và nhận xét mà nhóm đã thực hiện. Bên cạnh đó là những vấn đề tồn đọng mà nhóm gặp phải và chưa có hướng để giải quyết. Những ý tưởng và hướng phát triển trong tương lai cũng được đề cập đến trong phần cuối của báo cáo này.

1.3 Phạm vi

a) Mô tả khái quát phần mềm

Phần mềm AIMS (An Internet Media Store) là một phần mềm viết bằng Java giúp người dùng tìm kiếm và mua các loại mặt hàng liên quan đến các vật phẩm truyền thông đa phương tiện như sách, đĩa CD, đĩa DVD, …

b) Các chức năng chính

AIMS có mục tiêu tạo ra một trải nghiệm mua sắm trực tuyến thuận tiện, đáng tin cậy và đa dạng cho người dùng. Các tính năng chính của AIMS bao gồm:

- Theo dõi và tham quan cửa hàng trực tuyến, tìm kiếm và lọc ra các loại mặt hàng truyền thông đa phương tiện có trong hệ thống.

- Đăng ký tài khoản và đăng nhập vào hệ thống.

- Quản lý giỏ hàng cá nhân: bao gồm các hoạt động thêm sản phẩm vào giỏ hàng hoặc xóa sản phẩm khỏi giỏ hàng.

- Mua sản phẩm và thanh toán đơn hàng.

c) Cấu trúc thư mục mã nguồn

Phần mềm được viết bằng ngôn ngữ lập trình Java trên giao diện của JavaFX. Phần mềm được triển khai dựa trên kiến trúc của mô hình MVC. Ngoài các thư mục liên quan đến thư viện (lib), tài nguyên (resources) và kiểm thử (test) thì kiến trúc thư mục mã nguồn của phần mềm gồm các package chính như sau:

- Package common: chứa các file mã nguồn với mục đích sử dụng cho việc dùng chung ở nhiều nơi như các ngoại lệ (exception) và các interface chức năng (như Observer, Observable, …)

- Package controller: chứa các file mã nguồn của các controller được sử dụng trong hệ thống

- Package dao: chứa các file thực hiện nhiệm vụ truy xuất tới dữ liệu local sử dụng thư viện SQLite.

- Package entity: chứa các file định nghĩa ra các model sử dụng bên trong hệ thống

- Package subsystem: một hệ thống con chứa giao diện interbank hỗ trợ giao dịch liên kết ngân hàng thông qua các API

- Package utils: chứa các file công cụ chức năng được sử dụng nhiều nơi khác nhau

- Package views: là nơi định nghĩa ra các trang, giao diện người dùng và các màn hình của hệ thống

d) Các yêu cầu thêm cần cân nhắc cùng quá trình tái cấu trúc

Các yêu cầu thêm cần cân nhắc cùng quá trình tái cấu trúc bao gồm:

- Thêm một mặt hàng mới: Audio Book (Sách nói)
- Thêm màn hình xem chi tiết sản phẩm
- Thay đổi yêu cầu khi load giao diện
- Thêm cách tính khoảng cách, sử dụng thư viện mới
- Thêm phương thức thanh toán mới: Thẻ nội địa (Domestic Card)
- Thay đổi công thức tính phí vận chuyển
- Cập nhật lại chức năng hủy đơn hàng

Để đạt được mục tiêu trên, các hoạt động cần thực hiện trên mã nguồn bao gồm:

- Review codebase. Xem xét và đánh giá lại mã nguồn
- Refactor codebase. Tái cấu trúc lại mã nguồn
- Optimization. Chọn mô hình và phương pháp tái cấu trúc cho lợi ích tối ưu
- Testing. Kiểm thử lại các tính năng của dự án sau khi tái cấu trúc
- Scalability. Mở rộng dự án

e) Kết quả dự kiến

Kết quả dự kiến của dự án là sẽ làm sạch được mã nguồn của hệ thống AIMS. Mã nguồn sẽ trở nên khoa học, dễ hiểu, dễ chỉnh sửa và mở rộng cho các yêu cầu được thêm vào trong tương lai.

1.4 Danh sách thuật ngữ

1.5 Danh sách tham khảo

2 Đánh giá thiết kế cũ

2.1 Nhận xét chung

Nhìn chung mã nguồn của case study hiện tại đang ở mức chạy được, chưa phải là mã nguồn tốt. Về hiệu năng ứng biến của code so với các kế hoạch update trong tương lai, có thể nói khá là khó khăn và có rất nhiều vấn đề gặp phải nếu muốn update thêm các tính năng theo yêu cầu. Vì vậy hiệu năng ứng biến của mã nguồn hiện tại là không cao.

Vẫn còn rất nhiều những vấn đề mà mã nguồn hiện tại của case study gặp phải, như các vấn đề về cohesion và coupling, các nguyên lý SOLID, vấn đề clean code, … Nhìn chung thì mã nguồn cần được sửa đổi, tái cấu trúc lại và khắc phục những vấn đề trên để trở nên tốt hơn, dễ hiểu hơn và dễ dàng bảo trì, mở rộng hơn trong tương lai.

2.2 Đánh giá các mức độ coupling và cohesion

Trong thiết kế lập trình, để đạt được thiết kế tốt, chúng ta phải hướng tới ***loose coupling*** *và* ***high cohesion***.

- **Coupling** nói một cách đơn giản nó là sự phụ thuộc lẫn nhau giữa các modules (có thể hiểu là class, thành phần của phần mềm), là tính liên kết giữa các modules. Thiết kế code tốt thì phải theo hướng loose coupling tức là sự liên kết giữa các class, các thành phần trong phần mềm càng ít chặt chẽ càng tốt. Bởi vì nếu các thành phần quá phụ thuộc lẫn nhau, khi chúng ta thay đổi một thành phần sẽ ảnh hưởng tới rất nhiều thành phần liên quan khác.
- **Cohesion** cũng là chỉ mức độ về sự liên kết như coupling nhưng nó là sự liên kết ở mức chi tiết. Sự liên kết giữa các thành phần bên trong một module càng chặt chẽ thì thiết kế đó càng tốt tức là high cohesion. Lý do cho high cohesion là tất cả các yếu tố liên quan đến nhau được đặt trong một module để hướng tới một nhiệm vụ nhất định.

Hiện tại, trong code base vẫn còn tồn lại nhiều class vi phạm các mức độ của coupling và cohesion. Có vài class chỉ vi phạm ở mức độ nhỏ (Như Stamp coupling, sequential cohesion,...), tuy nhiên có nhiều class đang vi phạm coupling

và cohesion ở mức độ cao hơn, đòi hỏi cần áp dụng các biện pháp để khắc phục triệt để.

2.2.1 Coupling

#	*Các mức độ về Coupling*	*Module*	*Mô tả*	*Lý do*
1	Content	entity.cart (Cart)	getListMedia()	Trả về trực tiếp tham chiếu List<CartItem> lstCartItem của lớp thay vì trả về bản copy
2	Content	entity.order (Order)	getOrderMediaList()	Trả về trực tiếp tham chiếu List orderMediaList của lớp thay vì trả về bản copy
3	Common	controller(SessionInfomation.java)	Nhiều class sử dụng chung các trường và method của SessionInformation	Các class có sử dụng chung: AuthenticationController.java, BaseController.java, PaymentController.java, ViewCartController.java.
4	Common	entity.shipping (Shipping Configs)	Nhiều class sử dụng chung các trường và method của ShippingConfigs	Các class có sử dụng chung: ShippingScreenHandler.java, ...
5	Control	Interbank PayloadConverter	hàm extractPaymentTransaction() có phần xử lý để throw ra exception vi phạm	Dùng switch case với từng mã lỗi và xử lý tương ứng, trong khi hàm này có trách nhiệm khác
6	Stamp	MediaDAO.java (updateMediaFieldById), Cart.java	Vi phạm stamp coupling ở nhiều nơi	Nhiều tham số truyền vào thừa mà không sử dụng

		(checkMeddiaInCart.java), CartItem.java (Constructor CartItem), ...		
7	Data		Hiện có nhiều lớp có mức tốt là data coupling trong codebase	

2.2.2 Cohesion

#	***Các mức độ về Cohesion***	***Module***	***Mô tả***	***Lý do***
1	Coincidental	utils (Utils.java)	DATE_FORMATTER, LOGGER	Trong lớp Utils đang thực hiện quá nhiều công việc không liên quan đến nhau (God class)
2	Coincidental	views.screen (ViewsConfig)	Các thuộc tính đường dẫn PATH,PERCENT_VAT, REGULAR_FONT, getCurrencyFormat()	Trong lớp ViewsConfig chứa nhiều nhóm thuộc tính, phương thức không liên quan đến nhau (God class)
3	Logical	views.screen.popup (PopupScreen)	Các method success(), error(), loading()	Các method success(), error(), loading() đều là xử lý các dạng popup khác nhau nhưng thật sự không liên quan tới nhau

4	Temporal	views.screen (Các lớp ScreenHandler)	setupData() và setupFunctionality() vi phạm temporal	Hai method setupData() và setupFunctionality() được thiết lập chạy theo thứ tự khi bắt đầu khởi tạo các màn hình
5	Procedural	Controller(PaymentController)	Các method getExpirationDate(), payOrder()	Hai method trên thực hiện theo tuần tự cụ thể (Kiểm tra ngày hết hạn thẻ, rồi mới trả tiền order)
6	Communicational	Controller(AuthenticationController.java)	Các method isAnonymousSession, getMainUser, login, logout vi phạm	isAnonymousSession, getMainUser, login, logout thực hiện chức năng có liên quan tới SessionInformation
7	Communicational	Controller(BaseController.java)	Các method checkMediaInCart và getListCartMedia vi phạm	Các method trong class đều làm việc với SessionInformation.cartInstance
8	Communicational	Controller(ViewCartController.java)	Các method checkAvailabilityOfProduct() và getCartSubtotal() vi phạm	Các method checkAvailabilityOfProduct() và getCartSubtotal() cùng hoạt động trên SessionInformation.cartInstance
9	Communicational	dao.media (BookDao)	Các method getMediaById vi phạm	Các method của BookDao hoạt động trên data có dạng Book (Thuộc entity.media)
10	Communicationa	dao.media (CDDao)	Các method getMediaById vi phạm	Các method của CDDao hoạt động trên data có dạng CD (Thuộc entity.media)
11	Communicationa	dao.media (DVDDao)	Các method getMediaById vi phạm	Các method của DVDDao hoạt động trên data có dạng DVD (Thuộc entity.media)

12	Communicational	subsystem.interbank(InterbankBoundary.java)	Trả về của hàm query vi phạm cohesion	Hàm query trả về kiểu dữ liệu có liên quan tới class ApplicationProgrammingInterface	
13	Sequential	views.screen.invoice(MediaInvoiceScreenHandler.java)	setOrderItem và setMediaInfo vi phạm Sequential	setOrderItem sẽ thiết lập orderItem trong đối tượng, sau đó gọi setMediaInfo xử lý dựa trên giá trị đó	

2.3 Đánh giá việc tuân theo SOLID

2.3.1 SRP

Nhận xét: Trong source code của dự án, đang tồn tại nhiều class đảm nhiệm nhiều vai trò khác nhau cùng một lúc. Điều này sẽ gây nhiều khó khăn trong việc xác định chính xác tính năng cần chỉnh sửa, hay cập nhật sau này nằm ở đâu, đồng thời gây rối và làm phức tạp không cần thiết cho source code.

#	***Module***	***Mô tả***	***Lý do***
1	entity.cart (Cart.js)	Class Cart đang nắm nhiều trách nhiệm	Class cart vừa đảm nhiệm chức năng quản lý cart thông thường (thêm, gỡ bỏ media,..), vừa có trách nhiệm thông báo tình hình cart (Có trống hay không, số lượng các media trong cart, tổng tiền, hay media có đang bán hay không), việc này có thể giao cho ViewCartController quản lý.

2	entity.shipping(DeliveryInfo.java)	Class DeliveryInfo đang nắm 2 trách nhiệm	Tại thông tin khoảng cách, thì class nên tính luôn thông tin phí khoảng cách (Dựa trên thông tin address và province, rồi bind vào property distanceFee). Việc tạo phương thức calculateShippingFee khiến cho deliveryInfo đóng vai trò vừa lưu giữ thông tin vận chuyển, vừa cung cấp công cụ tính giá shipping.
	subsystem.interbank (InterbankPayloadConverter.java)	Class đóng vai trò convert interbank payload, nhưng có hàm getToday (Lấy ngày hôm nay), không hợp nhất với mục đích của class	Class có chứa hàm getToday, hàm này khá ngắn và chỉ được dùng duy nhất chỉ trong class này, nên được gộp chung với convertToRequestPayload
	Utils (ApplicationProgrammingInterface.java)	Class đóng vai trò xử lý nhiều HTTP method khác nhau	Class đóng vai trò xử lý nhiều HTTP method khác nhau (Get, Post). Việc xử lý hai method get và post nên được tách riêng ra
	utils (Ultil.java)	Class đóng vai trò god class (Xử lý nhiều các tác vụ không liên quan)	Lớp này là GodClass khi nhận đa trách nhiệm :DateFormat và Logger
	views.screen.popup (PopupScreen.java)	Class đảm nhiệm nhiều trách nhiệm như xử lý nhiều loại popup không giống nhau	Ba hàm success, error, loading có 3 trách nhiệm khác nhau tương ứng với các loại popup khác nhau
	views.screen (ViewsConfig.java)	Class đang nắm nhiều trách nhiệm	Class đang chứa các config về PATH, VAT, FONT vốn không liên quan tới nhau

2.3.2 OCP

Nhận xét: Source code vẫn tồn tại vài class vi phạm OCP. Điều này có thể gây khó khăn cho việc cập nhật các tính năng mới sau này.

#	*Module*	*Mô tả*	*Lý do*
1	controller (PlaceOrderController)	Các hàm validate nếu có bổ sung trong tương tai sẽ phải chỉnh sửa lại hàm validate tổng trong class	Các hàm validate nếu có bổ sung trong tương tai sẽ phải chỉnh sửa lại hàm validate tổng trong class, chúng nên được tách riêng ra thành class riêng biệt
2	views.screen.popup (PopupScreen.java)	Các chức năng liên quan tới các loại popup riêng đang được gộp chung trong một class	Các popup như loading, error. Success đang gộp chung trong một file, điều này gây khó khăn cho việc mở rộng thêm một loại popup mới sau này
3	utils (ApplicationProgrammingInterface)	Lớp API này đang chứa hai phương thức cố định là get và post	Mỗi khi thêm một phương thức mới như patch, delete, … thì phải modify lại mã nguồn của lớp này. Vì thế các nghiệp vụ này nên được tách thành các lớp riêng biệt để có thể dễ dàng mở rộng

2.3.3 LSP

#	*Module*	*Mô tả*	*Lý do*
1	dao.media (MediaDAO, BookDAO, CDDAO, DVDDAO)	3 lớp BookDAO, CD DAO, DVDDAO đang cùng kế thừa lớp MediaDAO. Lớp MediaDAO dùng để thao tác truy cập lên bảng Media nhưng các lớp kia lại truy cập lên các bảng cụ thể hơn → 3 lớp còn lại vi phạm LSP	Lớp MediaDAO không có lý do gì để làm cha của 3 lớp còn lại. Trong khi phương thức getAllMedia của MediaDAO trả về tất cả các Media thì getAllMedia của các lớp con chỉ trả về danh sách Media của mỗi lớp con ấy. Tương tự với các phương thức khác, các lớp con đã làm thay đổi hành vi của lớp cha và có thể sẽ

			làm chương trình chạy không đúng kết quả như đối với lớp cha nữa.

2.3.4 ISP

Hiện mã nguồn chưa vi phạm nguyên lý này.

2.3.5 DIP

#	*Module*	*Mô tả*	*Lý do*
1	view.screen (BaseScreenHandler)	Lớp BaseScreen Handler đang có một thuộc tính là Home ScreenHandler và lớp HomeScreenHa ndler lại đang là con của lớp BaseScreen Handler.	Lớp BaseScreenHandler vi phạm nguyên lý DIP khi đang phụ thuộc vào lớp có mức độ trừu tượng thấp hơn
2	Các module đang sử dụng trực tiếp lớp CreditCard như: Payment Controller, Int erbankInterfac e, Interbank Subsystem, …	Lớp PaymentControl ler có một thuộc tính thuộc kiểu CreditCar d, còn các lớp, interf ace khác sử dụng CreditCard như là tham số trong các phương thức	Lớp CreditCard là một lớp có mức độ trừu tượng thấp hơn các lớp khác. Vì thế nếu có một loại thẻ mới cần thêm vào hệ thống thì sẽ không thực hiện được đối với code base hiện tại

2.4 Các vấn đề về Clean Code

2.4.1 Clear Name

- PaymentController.java: Line 48 - Biến strs không có ý nghĩa gì, khiến code khó hiểu
- ViewCartController.java: Tên class không nên là động từ, nên chuyển thành CartController.java
- MediaDAO.java:

    - Line 20: Biến medium đại diện cho danh sách các Media, nhưng có ý nghĩa không hề liên quan
    - Line 54: Tham số tbname không thể hiện được ý nghĩa
- Cart.java:
    - Line 36, 45: Iterator obj không thể hiện rõ ràng ý nghĩa, nên đổi thành cartItemObj
    - Line 18, 22, 37, 46: Biến cm không phải cách viết tắt có thể tạo ra liên tưởng tới CartItem, nên chọn cách viết tắt khác hoặc viết rõ hẳn ra
    - Line 54: Iterator object không thể hiện rõ ràng ý nghĩa, nên đổi thành cartItemObj
- AIMSDB.java: Line 14 - Biến connect là một động từ, nên thay bằng connection
- Media.java: Line 49 - Thuộc tính updated_quantity viết theo snake_case, nên viết theo camelCase
- Order.java: Line 29 - Iterator object không thể hiện rõ ràng ý nghĩa, nên đổi thành cartItemObj
- InterbankSubsystem.java: Line 19 - Thuộc tính ctrl không thể hiện được ý nghĩa rõ ràng, nên chuyển thành interbankController
- ApplicationProgrammingInterface.java:
    - Line 31, 52: Biến in ý nghĩa không rõ ràng, có thể đổi thành reader
    - Line 33: Biến respone sai chính tả
- MyMap.java:
    - Tên class MyMap cũng như phương thức tương đối chung chung, không thể hiện được ý nghĩa.
    - Line 26: Biến max không có ý nghĩa rõ ràng
    - Line 31: Iterator it (nó) dễ gây hiểu lầm khi đọc code, nên đổi thành iter
- CartScreen.java: Line 72 - Biến im không phải cách viết tắt thông dụng của image, nên chuyển thành img
- MediaHandler.java: Line78 - Biến im không phải cách viết tắt thông dụng của image, nên chuyển thành img
- HomeScreenHandler.java:
    - Line 101: Biến object không thể hiện được ý nghĩa, nên đổi thành mediaObj
    - Line 103: Biến m không thể hiện được ý nghĩa, nên đổi thành mediaHandler
- BaseScreenHandler.java: Line 50 - Tham số string không có ý nghĩa, nên đổi thành screenTitle

2.4.2 Clean Function/Method

- PlaceOrderController.java (Line 78): Bỏ line này hoặc thay bằng LOGGER.info().
- CartScreenHandler.java (Line 110): Nên sử dụng hàm isEmpty().
- Các biến config trong file ShippingConfig.java nên set là final.
- ApplicationProgrammingInterface.java, Line 38, 39: Method substring trả về là string, nên toString() đi sau là không cần thiết.
- MyMap.java, line 38: Key là string, nên toString đi sau là không cần thiết. Đồng thời cụm từ "`Cannot resolve the input`" xuất hiện 12 lần, nên đặt là biến riêng.
- MyMap.java có phương thức toMyMap() quá phức tạp (Nhiều condition, try catch).
- Book.java, CD.java, DVD.java: Constructor có hơn 7 parameter, phức tạp hơn so với mức khuyến cáo (7).

2.4.3 Clean Class

- Class Utils ban đầu là một class rất chung chung và thực hiện nhiều nhiệm vụ khác nhau, có thể trở thành God class, cần phải tách ra
- Class ViewsConfig có nhiều hơn một lý do để thay đổi vì nó chứa: nhóm các thuộc tính Path, REGULAR_FONT, PERCENT_VAT, getCurrencyFormat(int num), cần phải tách ra
- Class PlaceOrderController ban đầu chứa cả các hàm validateDeliveryInfo, PhoneNumber, Address, Name không phải trách nhiệm của nó

2.5 Các vấn đề khác

<Ngoài các vấn đề bám theo nội dung lý thuyết kể trên, các vấn đề khác được liệt kê ở đây, theo cấu trúc bảng tương tự. Cần làm rõ các vấn đề cần xem xét trước khi đưa ra ví dụ.>

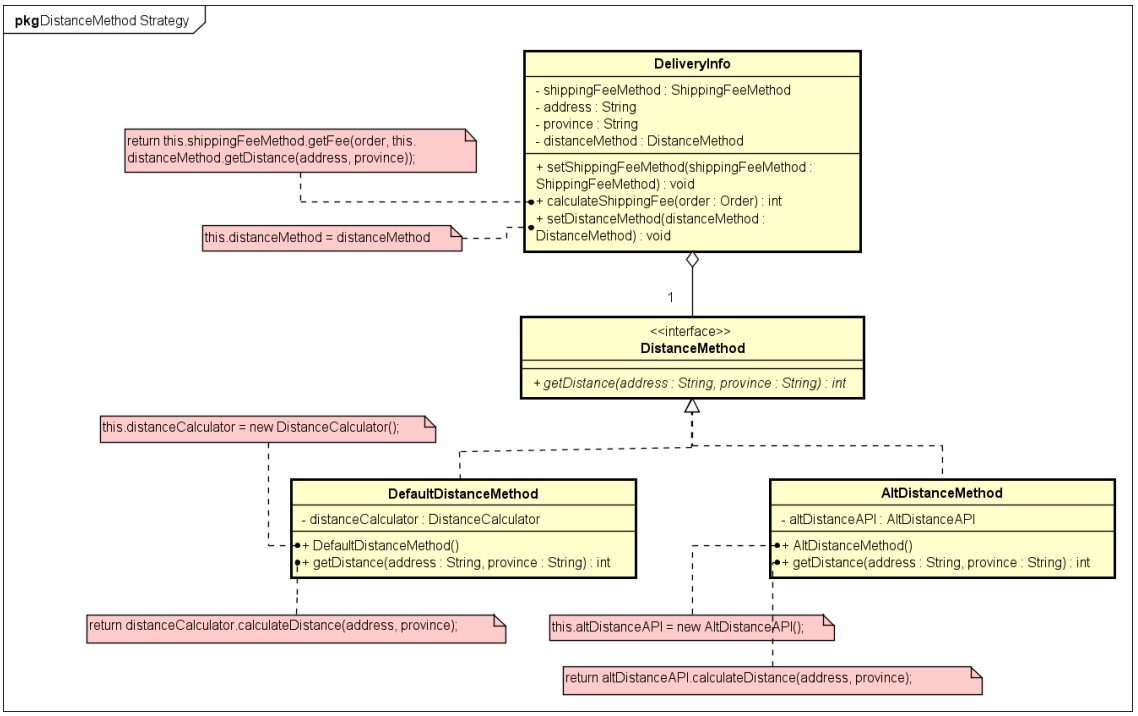
3 Đề xuất cải tiến

3.1. Vấn đề thay đổi thư viện DistanceCalculator và tương thích với codebase.

Mô tả. Trong codebase, thư viện DistanceCalculator cung cấp cho ta phương thức caculateDistance để giúp tính khoảng cách.Trong tương lai, dự án AIMS có thể sẽ hướng tới thay thế thư viện DistanceCalculator, với cách thức tính khoảng cách mới trong giai đoạn tạo DeliveryInfo.

Giải pháp. Trong codebase hai thư viện, do phương thức tính khoảng cách (caculateDistance) ở hai bên khác parameter đầu vào, nên ta sử dụng Adapter Patern, tạo nên interface chung là DistanceMethod, và 2 class con sẽ implement interface này là DefaultDistanceMethod (Chứa cách tính khoảng cách cũ), và AltDistanceMethod (Chứa cách tính khoảng cách mới), kéo về dạng chung caculateDistance(String address, String province), để đảm bảo sự tương thích. Đồng thời áp dụng thêm Strategy patern, bằng cách này, ta có thể lựa chọn sử dụng cách tính mới hoặc quay trở lại cách tính cũ nếu cần.

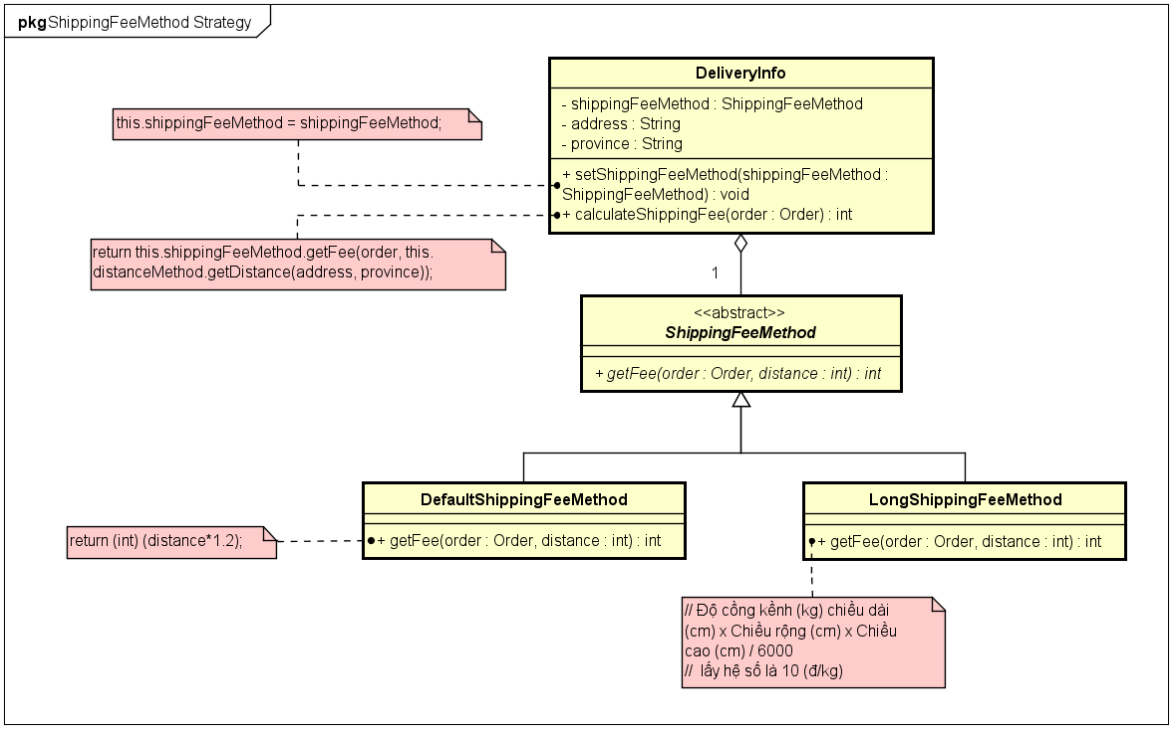
- DefaultDistanceMethod dùng thư viện CalculateDistance.
- AltDistanceMethod dùng thư viện AltDistanceAPI.



3.2. Vấn đề về chỉnh sửa hệ số tính ShippingFee

Mô tả. Hệ số nhân tính phí vận chuyển đang được đặt giá trị cố định là 1,2. Nếu tương lai ta muốn thay đổi hệ số nhân thì cần phải sửa lại code base.

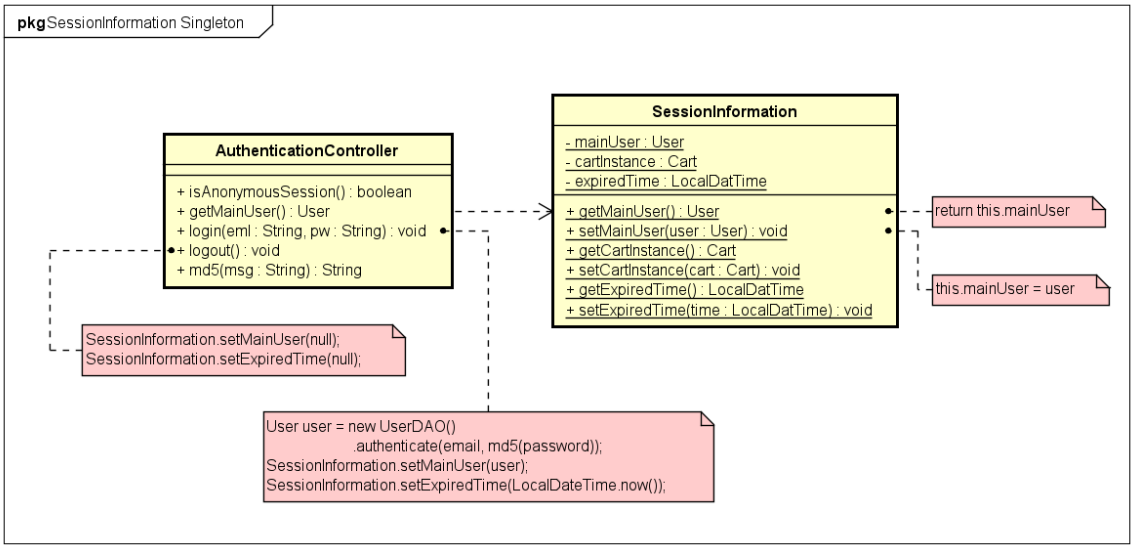
Giải pháp. Dùng Strategy Patern, tạo abstract class ShippingFeeMethod với phương thức trừu tượng getFee(Order order, int distance). Với mỗi cách tính giá ship khác nhau, ta tạo một class kế thừa lại class ShippingFeeMethod và @Override phương thức getFee, trong đó có thể triển khai các logic khác nhau để tính giá ship. Cụ thể đối với class DefaultShippingFeeMethod, phương thức getFee sẽ trả về giá trị tiền ship bằng khoảng cách nhân 1,2. Nếu như trong tương lai, muốn áp dụng cách tính tiền ship khác, ta chỉ cần tạo thêm một class mới và kế thừa lại ShippingFeeMethod.



3.3. Vấn đề Common Coupling của SessionInformation

Mô tả. Các biến mainUser, cartInstance và expiredTime của lớp Session Information đang là các biến toàn cục (public static) và được truy cập trực tiếp từ rất nhiều nơi trong hệ thống. Điều này là vi phạm Common Coupling.

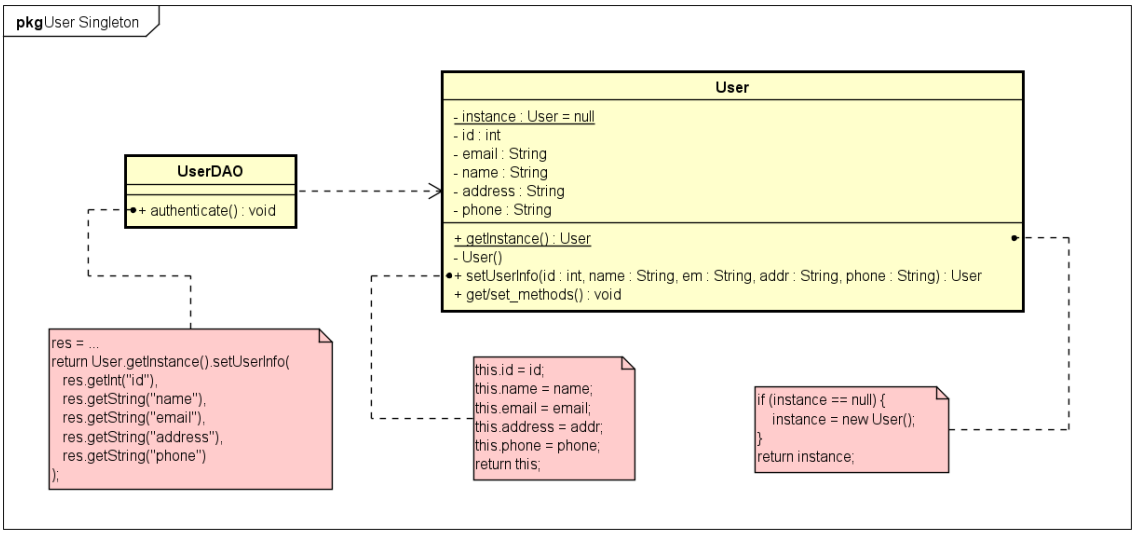
Giải pháp. Chuyển chỉ định truy cập của các biến toàn cục trên thành private và kiểm soát truy cập bằng cặp phương thức getter và setter cho mỗi biến thành phần.



3.4. Vấn đề sự tồn tại duy nhất và toàn cục của thông tin phiên người dùng trong hệ thống

Mô tả. Thông tin về phiên làm việc của người dùng sau khi đã đăng nhập vào hệ thống phải là toàn cục (có thể được truy cập tại bất cứ đâu) và phải là duy nhất (không thể có hai người dùng khác nhau cùng làm việc trên cùng một phiên đăng nhập, cùng một máy tính).

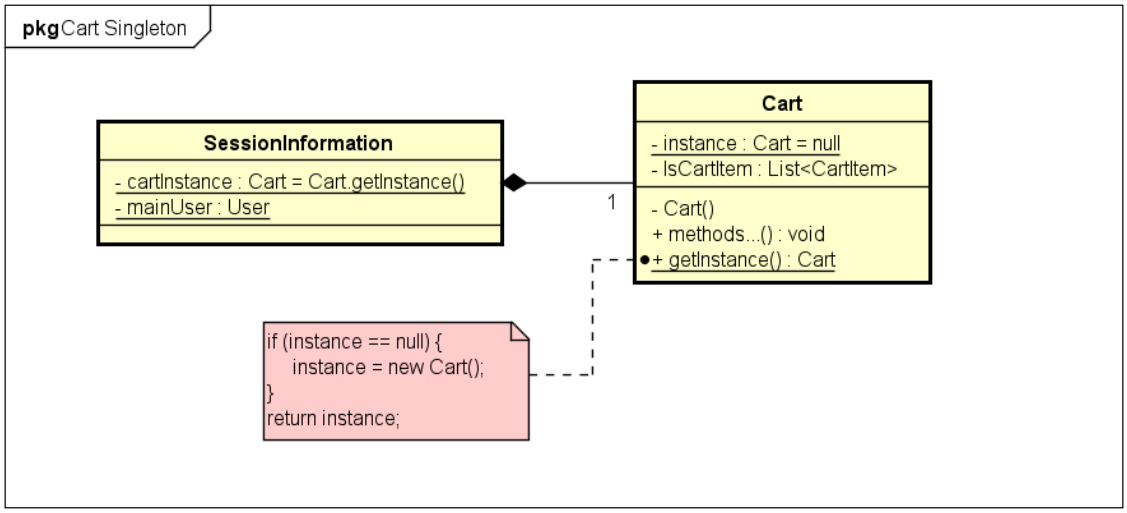
Giải pháp. Sử dụng mẫu thiết kế Singleton cho lớp User. Lớp UserDAO (1 trong những client) có thể sử dụng biến toàn cục cho thông tin phiên user này bằng cách gọi User.getInstance().



3.5. Vấn đề sự tồn tại duy nhất và toàn cục của giỏ hàng trong hệ thống

Mô tả. Thông tin về giỏ hàng của người dùng khi đã đăng nhập và sử dụng hệ thống phải là toàn cục để có thể lấy được các sản phẩm trong giỏ hàng ở bất cứ đâu và bất cứ khi nào. Giỏ hàng của người dùng cũng phải là duy nhất vì một người dùng không thể có hai giỏ hàng khi sử dụng hệ thống.

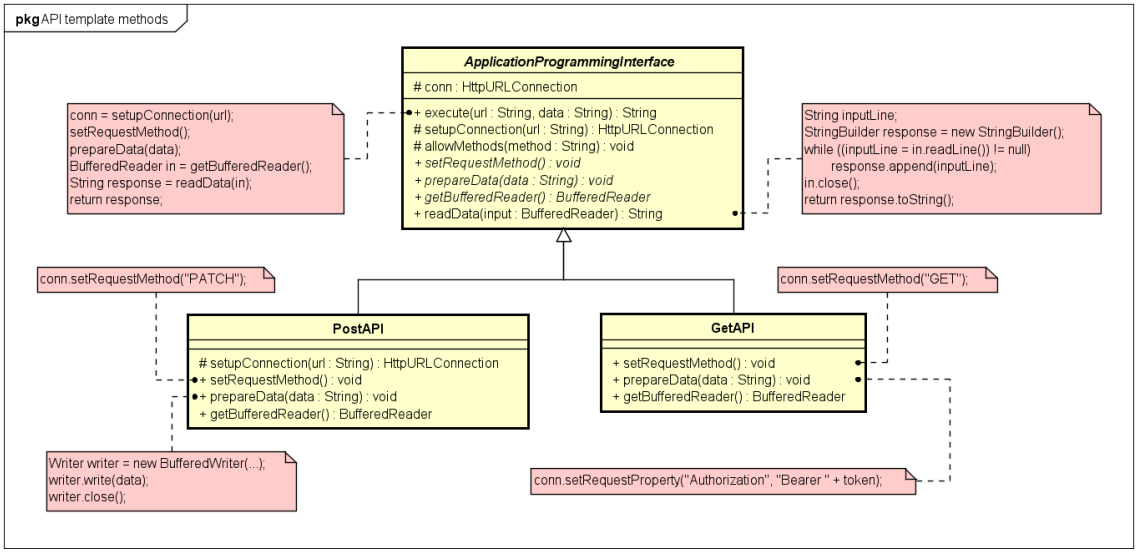
Giải pháp. Sử dụng mẫu thiết kế Singleton cho lớp Cart. Ở đây biến Cart được sử dụng làm thuộc tính trực tiếp và được khởi tạo ngay khi thông tin phiên được tạo ra bên trong lớp SessionInformation.



3.6. Vấn đề SRP và OCP trong lớp ApplicationProgramming Interface

Vấn đề. Lớp ApplicationProgrammingInterface đang thực hiện cả hai trách nhiệm là post và get. Lớp này cũng chỉ hỗ trợ hai phương thức này, chưa phù hợp cho việc thêm các phương thức mới trong tương lai.

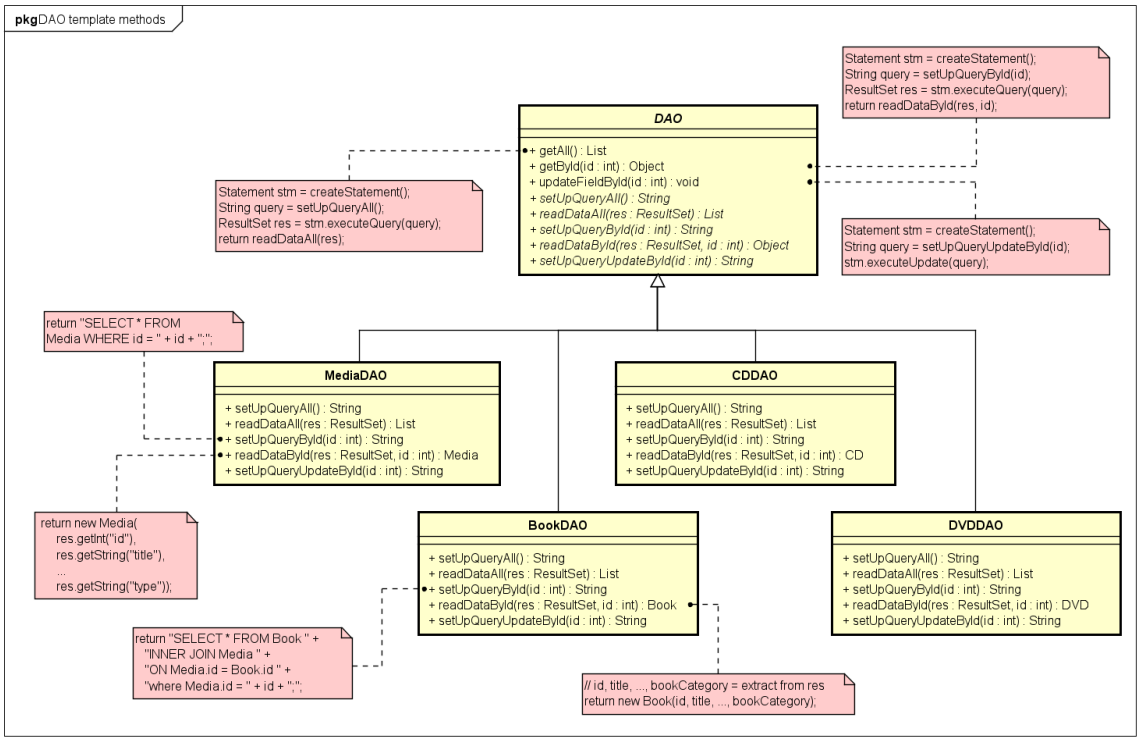
Giải pháp. Chuyển lớp API thành abstract và tách các nghiệp vụ post, get ra thành các lớp riêng cùng kế thừa abstract. Đồng thời sử dụng Design pattern Template Method cho method execute(). Các step trong template method này là: setupConnection, setRequestMethod, prepareData, getBufferedReader, readData.



3.7. Vấn đề LSP trong nhóm các lớp Media DAO.

Vấn đề. Lớp MediaDAO không có lý do gì để làm cha của 3 lớp còn lại. Trong khi phương thức getAllMedia của MediaDAO trả về tất cả các Media thì getAllMedia của các lớp con chỉ trả về danh sách Media của mỗi lớp con ấy. Tương tự với các phương thức khác, các lớp con đã làm thay đổi hành vi của lớp cha và có thể sẽ làm chương trình chạy không đúng kết quả như đối với lớp cha nữa.

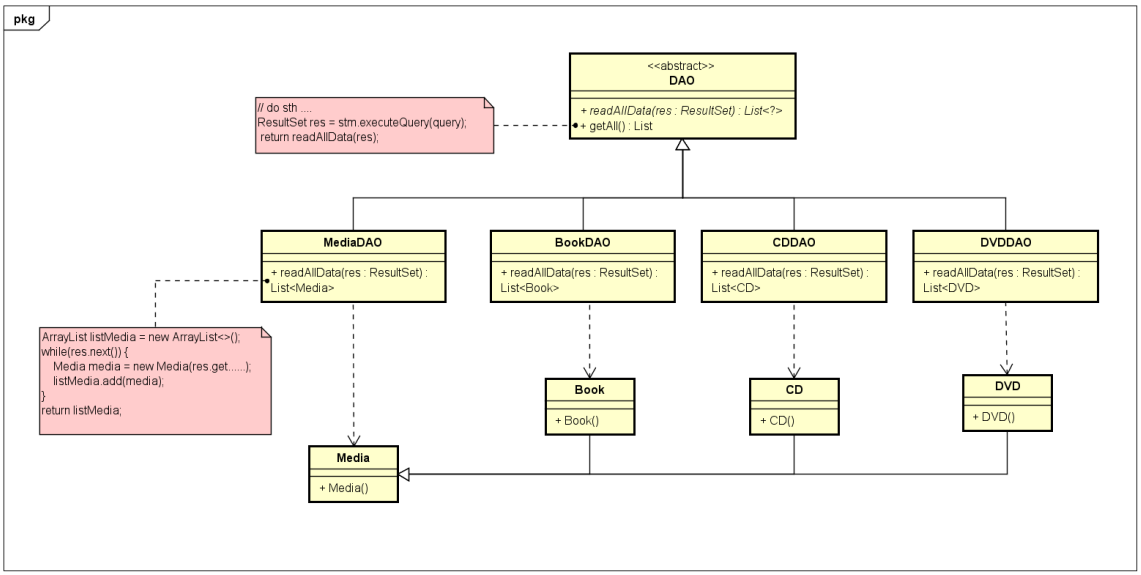
Giải pháp. Tái cấu trúc kế thừa các lớp, trong đó thành lập lớp trừu tượng DAO làm cha của 4 lớp MediaDAO, BookDAO, CDDAO và DVDDAO. Bên cạnh đó, sử dụng Design pattern Template Method cho 3 phương thức là getAll(), getById() và updateFieldById(). Các step trong mỗi template method như biểu đồ lớp phía dưới, cơ bản gồm: createStatement, lấy ra câu truy vấn SQL, thực thi câu truy vấn và đọc dữ liệu trả về nếu có. Khi có một loại media mới chỉ cần tạo lớp DAO tương ứng kế thừa lớp cha và ghi đè lại các phương thức step trong các template method yêu cầu.



3.8. Sử dụng factory method sau khi tái cấu trúc các lớp Media DAO

Mô tả. Sau khi tái cấu trúc lại, các lớp DAO sẽ có chung method getAll() sẽ thực hiện setUpQuery để truy cập database và sau đó gọi readAllData(res) trả ra danh sách mọi đối tượng thuộc loại Media tương ứng

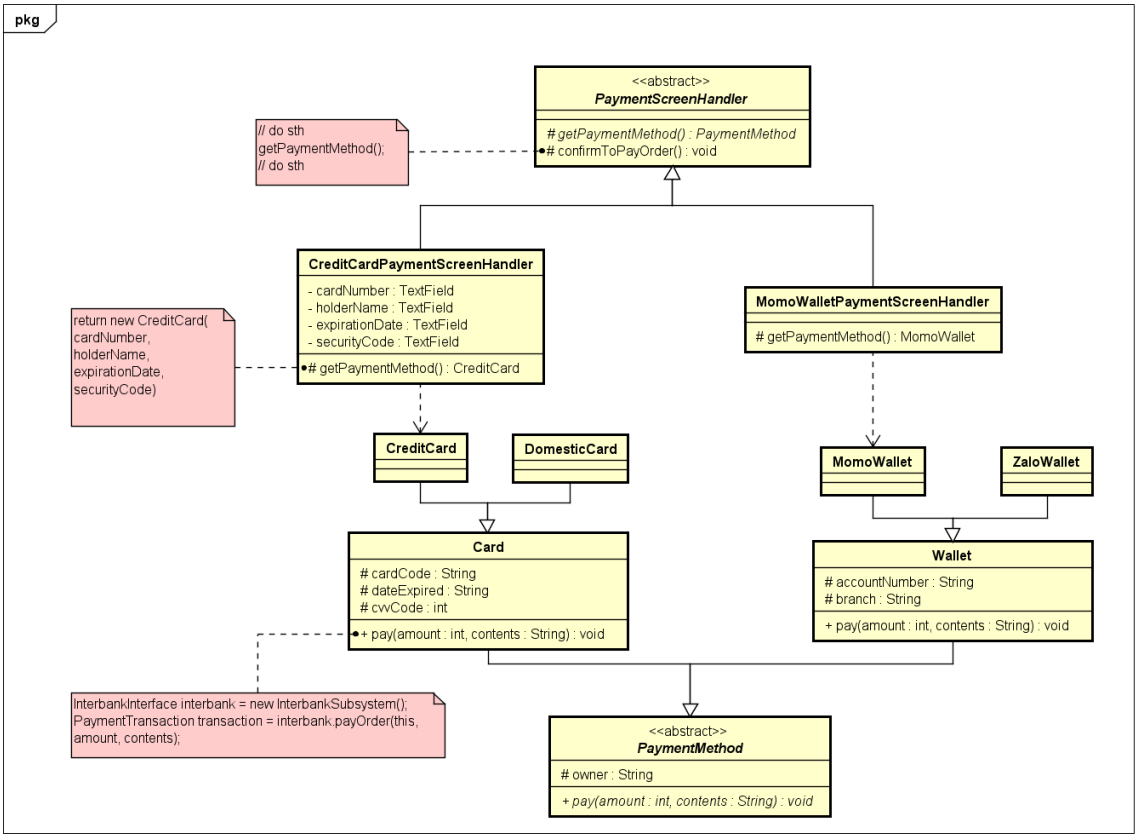
Giải pháp. Sử dụng Factory Method để khi sử dụng chỉ cần gọi phương thức readAllData(res) mà không quan tâm tới việc từng danh sách cho mỗi loại (Media, Book, CD, DVD) được tạo như nào.



3.9. Vấn đề DIP, OCP đối với các lớp ở mức cao phụ thuộc vào lớp cụ thể CreditCard

Mô tả. Các lớp PaymentController, PaymentTransaction, InterbankPayloadConverter, InterbankInterface (mức cao hơn) lại phụ thuộc vào lớp CreditCard (lớp cụ thể hơn). Trong tương lai khi mở rộng cơ chế thanh toán với nhiều loại thẻ và có thể có thêm ví thanh toán, sẽ phải modify lại rất nhiều ở các lớp đó.

Giải pháp. Tái cấu trúc lại cây thanh toán và màn hình sử dụng của từng loại, sử dụng factory method: getPaymentMethod() ở các lớp PaymentScreenHandler muốn chỉ cần gọi method lấy ra loại thanh thoán (có thể là thẻ tín dụng, thẻ quốc nội, ví điện tử, …) để thực hiện confirmPayOrder() hoặc các tác vụ liên quan mà không cần thiết phải thực hiện các thao tác tạo phức tạp của từng loại từ việc lấy String từ TextField sau đấy truyền những thông tin cần thiết để tạo ra instance cho loại hình thanh toán tương ứng.


pkg
<<abstract>>
PaymentScreenHandler
getPaymentMethod() : PaymentMethod
confirmToPayOrder() : void
// do sth
getPaymentMethod();
// do sth
CreditCardPaymentScreenHandler
- cardNumber : TextField
- holderName : TextField
- expirationDate : TextField
- securityCode : TextField
getPaymentMethod() : CreditCard
MomoWalletPaymentScreenHandler
getPaymentMethod() : MomoWallet
return new CreditCard(
cardNumber,
holderName,
expirationDate,
securityCode)
CreditCard
DomesticCard
MomoWallet
ZaloWallet
Card
cardCode : String
dateExpired : String
cvvCode : int
+ pay(amount : int, contents : String) : void
Wallet
accountNumber : String
branch : String
+ pay(amount : int, contents : String) : void
InterbankInterface interbank = new InterbankSubsystem();
PaymentTransaction transaction = interbank.payOrder(this,
amount, contents);
<<abstract>>
PaymentMethod
owner : String
+ pay(amount : int, contents : String) : void

4 Tổng kết

4.1 Kết quả tổng quan

Tổng kết lại, nhóm đã cải tiến được source code rất nhiều với việc phát hiện các vấn đề nhờ tính chất low coupling - high cohesion, nguyên lý SOLID, tái cấu trúc lại code base với các tính chất OOP và sử dụng các mẫu design patterns đã được học (singleton, template method, strategy, factory method, adapter, …). Về mặt hiệu năng thì không thấy được rõ ràng sự cải thiện, tuy nhiên nhìn về mặt source code có thể thấy được sự cải thiện rõ ràng, tối ưu hơn, mã nguồn được thiết kế có tính tái sử dụng hơn, an toàn hơn, dễ thích nghi với các yêu cầu thay đổi, nâng cấp trong tương lai hơn.

4.2 Các vấn đề tồn đọng

Mã nguồn vẫn còn nhiều nơi chưa được refactor hợp lý ví dụ như việc hiển thị popup cho các Error trong tương lai có thể chuyển thành các hiển thị khác, nhiều nơi tên các biến vẫn còn chưa được xem xét và sửa sao cho phù hợp, vẫn còn nhiều lỗi nhỏ chưa được clean. Có thể tiềm tàng nhiều trường hợp mà nhóm chưa nhận ra và có thể giải quyết bằng các design patterns khác ngoài các mẫu đã học. Cần phải giành nhiều thời gian hơn để xem xét chi tiết và kỹ càng hơn toàn bộ code base