ĐẠI HỌC BÁCH KHOA HÀ NỘI

Viện Công nghệ thông tin và Truyền thông

Báo cáo Mẫu thiết kế phần mềm

Version 1.0

Nhóm 10

Danh sách thành viên:

Nguyễn Bá Quân	20173315
Phạm Thế Tài	20173351
Vũ Văn Quân	20173313
Lê Xuân Quang	20173330

Hà Nội, tháng 06 năm 2021

Mục lục

1 Tổng quan

1.1 Mục tiêu

Mục đích của báo cáo là chỉ ra những vấn đề trong thiết kế của mã nguồn. Người sử dụng báo cáo là lập trình viên, qua báo cáo này họ có thể xác định các vấn đề trong thiết kế của mã nguồn và cải tiến thiết kế của mã nguồn. Thiết kế của mã nguồn sau khi cải tiến cần không vi phạm hoặc vi phạm ít nhất nguyên lý thiết kế SOLID, cùng với đó thiết kế cũng cần đạt được low coupling và high cohesion.

Nội dung báo cáo gồm 3 phần:

- Phần 1 Tổng quan: Trình bày khái quát về phần mềm, danh sách thuật ngữ và các tài liệu tham khảo.
- Phần 2 Đánh giá thiết kế cũ: Trình bày các kết quả thu được sau quá trình code review trên mã nguồn.
- Phần 3 Đề xuất cải tiến: Đưa ra những đề xuất cần cải tiến để khắc phục những vấn đề của mã nguồn.

1.2 Phạm vi

(1) Mô tả khái quát phần mềm:

AIMS Project là một hệ thống đa nền tảng hoạt động 24/7, cho phép người dùng mới có thể làm quen dễ dàng. Hệ thống này có thể cho phép phục vụ 1000 khách hàng cùng lúc mà hiệu suất không bị giảm đáng kể, đồng thời có thể hoạt động 300 giờ liên tục không hỏng hóc. Ngoài ra, hệ thống có thể hoạt động trở lại bình thường trong vòng 1 giờ sau khi xảy ra lỗi. Thời gian đáp ứng của hệ thống tối đa là 1 giây khi bình thường hoặc 2 giây lúc cao điểm.
Trong hệ thống thương mại điện tử AIMS Project, quản trị viên có thể thêm, xem, sửa, xóa bất kỳ sản phẩm nào. Tuy nhiên, quản trị viên chỉ có thể thêm hoặc sửa với một sản phẩm tại một thời điểm, nhưng lại có thể xóa tới 10 sản phẩm cùng một lúc. Ngoài ra, quản trị viên không thể xóa hoặc cập nhật quá 30 sản phẩm ví lý do bảo mật nhưng có thể thêm không giới hạn số sản phẩm trong một ngày.
Về phía khách hàng, khi khởi động, hệ thống sẽ hiện ra danh sách của 20 sản phẩm bất kỳ ở mỗi trang. Để tìm kiếm sản phẩm, khách hàng sử dụng các đặc tính của sản phẩm để tìm kiếm. Hệ thống sẽ hiện ra 20 sản phẩm liên quan trong mỗi trang tìm kiếm. Bên cạnh đó, khách hàng có thể sắp xếp sản phẩm theo giá cả hoặc có thể thêm sản phẩm với số lượng tương ứng vào giỏ hàng (cart) hiện tại.
Khi xem giỏ hàng, hệ thống sẽ hiện ra thông tin giỏ hàng, bao gồm tổng giá cả sản phẩm chưa bao gồm VAT, tổng giá cả sản phẩm đã bao gồm VAT, danh sách sản phẩm với thông tin sản phẩm (tên sản phẩm, số lượng, và giá

cả). Đồng thời, hệ thống cũng thông báo tới khách hàng nếu số lượng hàng tồn trong kho bất kỳ sản phẩm không đủ và sẽ hiện ra số lượng này của từng sản phẩm bị thiếu. Khi thay đổi ý định, khách hàng cũng có thể bỏ sản phẩm khỏi giỏ hàng. Ngoài ra, chỉ có 1 giỏ hàng trong mỗi phiên bản chạy của hệ thống, đồng thời, giỏ hàng sẽ được làm trống sau khi thanh toán đơn hàng thành công. Từ giao diện xem giỏ hàng này, khách hàng có thể yêu cầu đặt hàng.

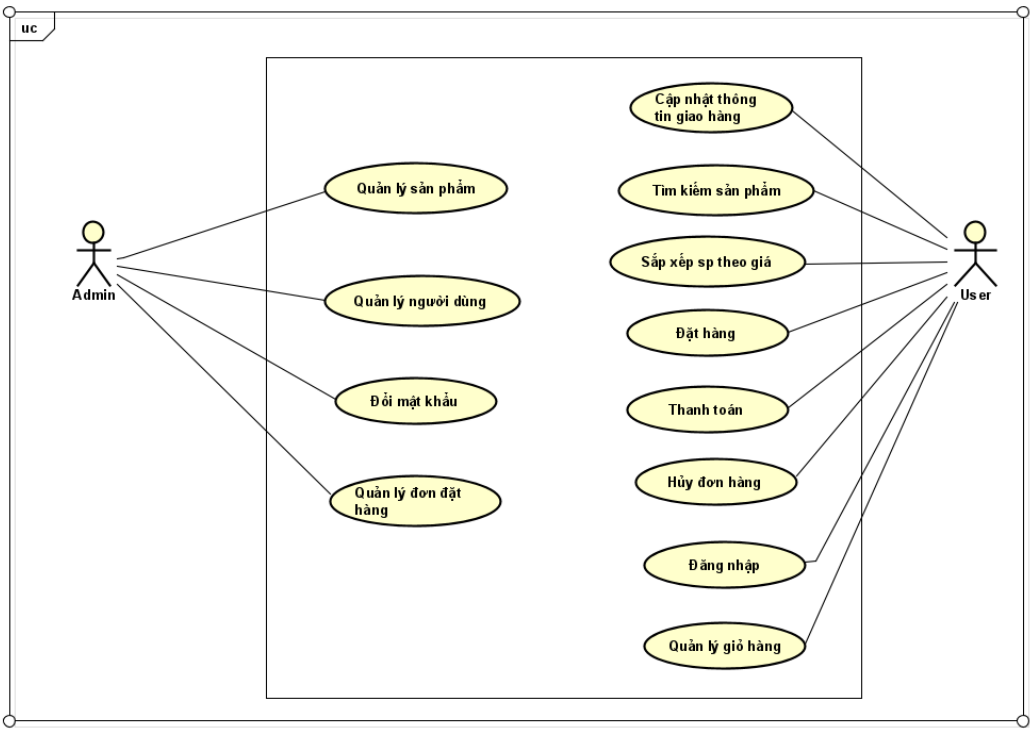
Trong AIMS Project, quá trình đặt hàng trải qua 5 bước chính: (1) đặt hàng (order placement), (2) thanh toán (order payment), (3) duyệt đơn hàng (order approval), (4) vận chuyển (delivery), và (5) hỗ trợ sau đặt hàng (post order support)

(2) Các chức năng chính của phần mềm

- Quản trị viên:
  - Quản lý sản phẩm
  - Quản lý người dùng
  - Quản lý đơn đặt hàng

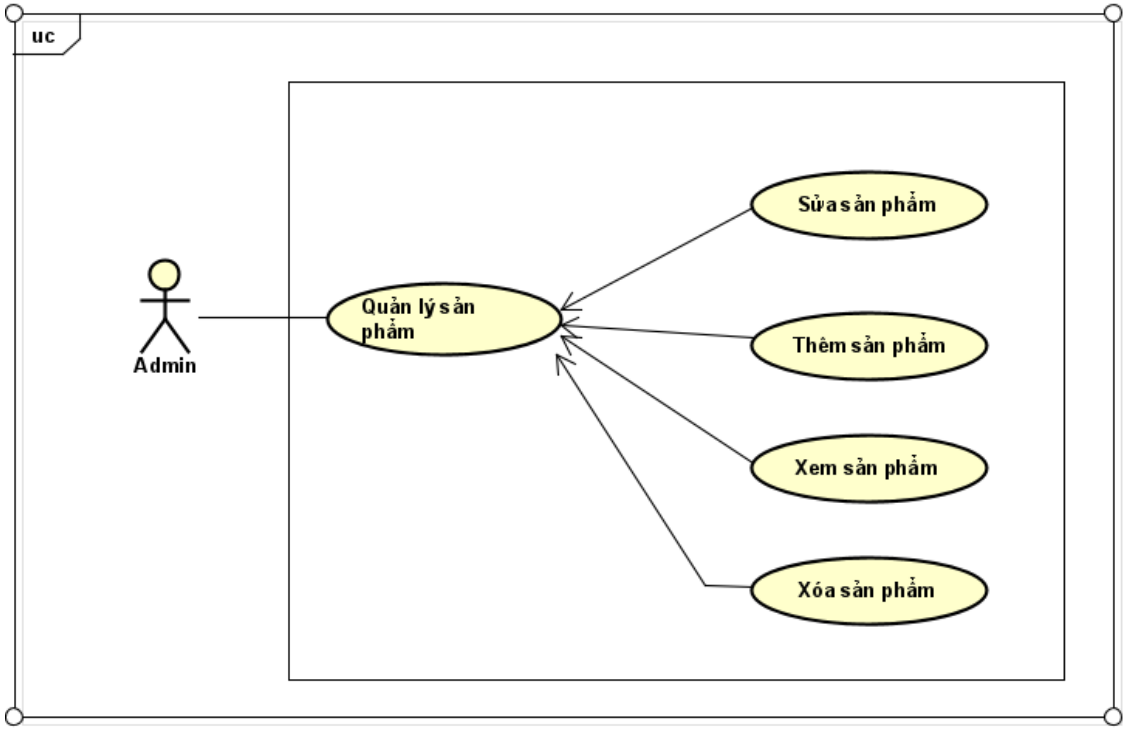
- Người dùng:
  - Tìm kiếm sản phẩm
  - Quản lý giỏ hàng
  - Đặt hàng
  - Quản lý thông tin đơn hàng
  - Thanh toán

2.1 Biểu đồ use case tổng quan hệ thống



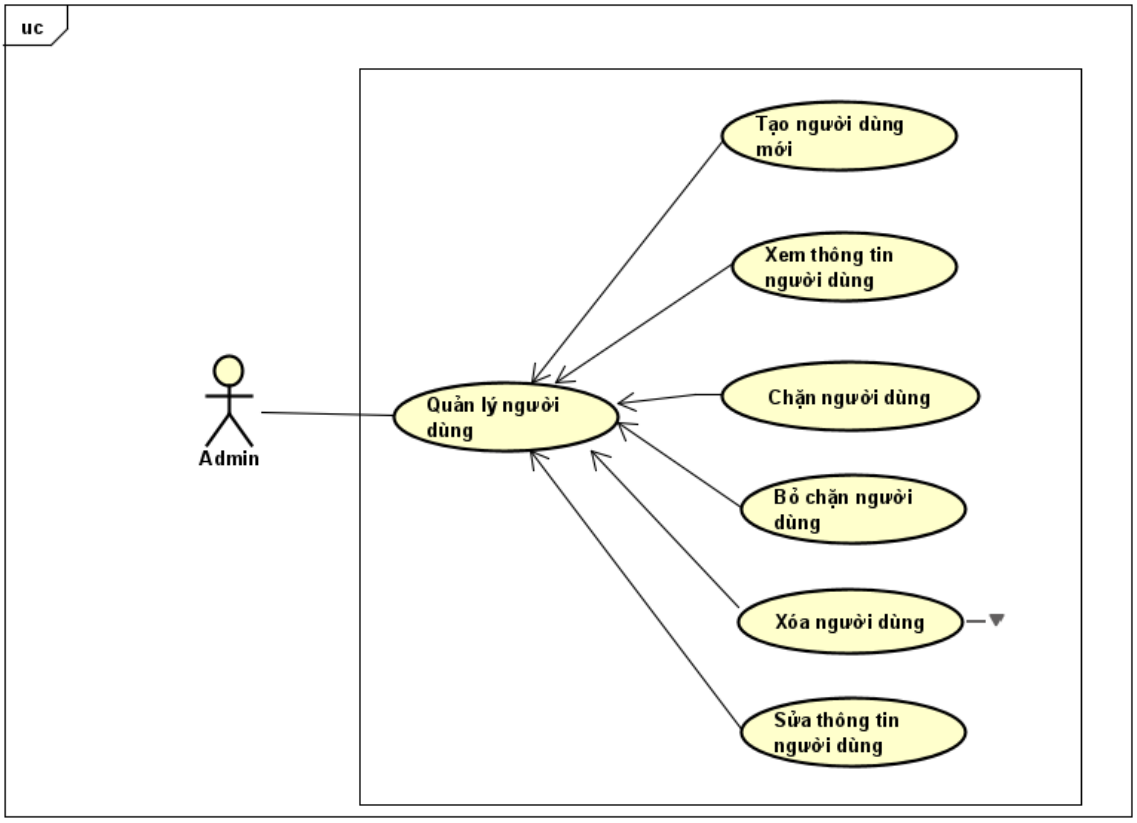
Hình 1.1 Biểu đồ use case tổng quan

2.2 Biểu đồ phân rã use case quản lý sản phẩm



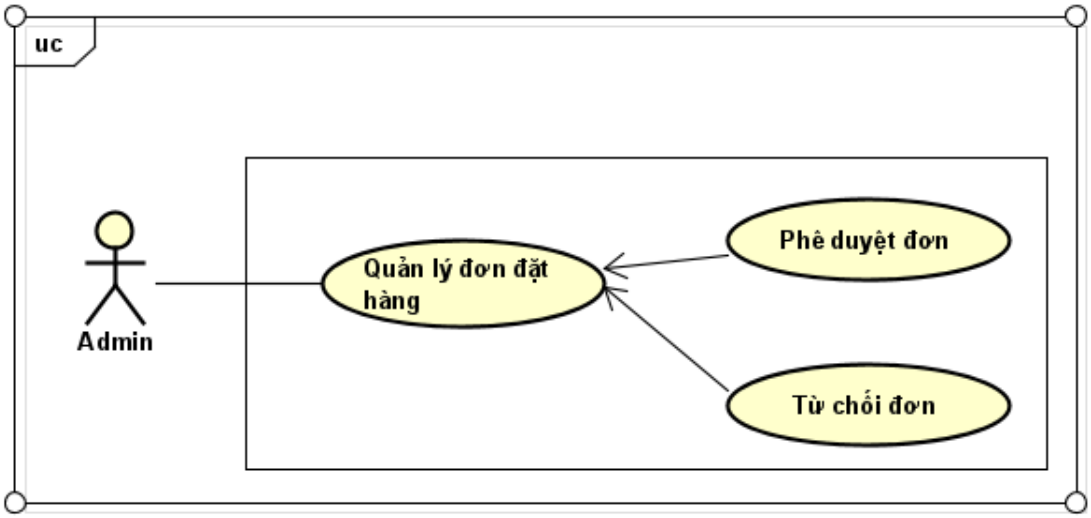
Hình 1.2 Biểu đồ phân rã use case quản lý sản phẩm

2.3 Biểu đồ phân rã use case quản lý người dùng



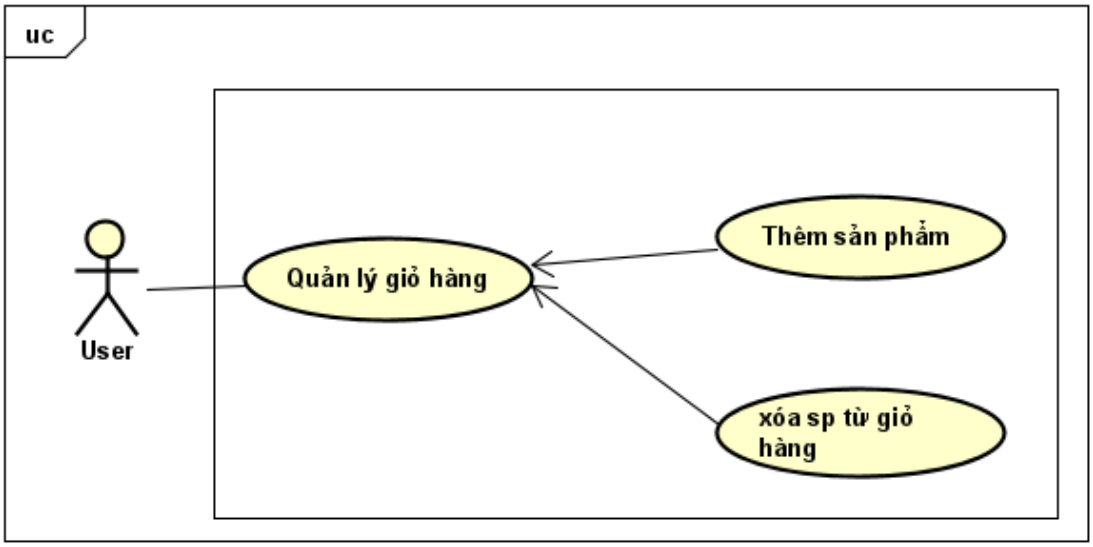
Hình 1.3 Biểu đồ phân rã use case quản lý người dùng

2.4 Biểu đồ phân rã use case quản lý đơn đặt hàng



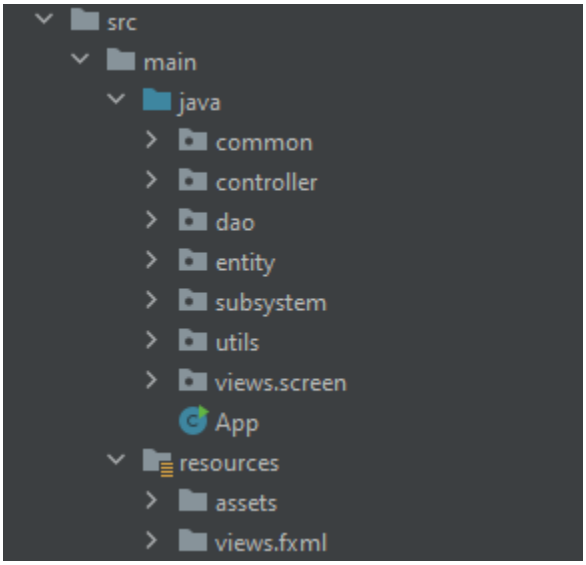
Hình 1.4 Biểu đồ phân rã use case quản lý đơn đặt hàng

2.5 Biểu đồ phân rã use case quản lý giỏ hàng



Hình 1.5 Biểu đồ phân rã use case quản lý giỏ hàng

(3) Cấu trúc mã nguồn



Hình 1.6 Cấu trúc mã nguồn

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

1. Thêm mặt hàng Media mới: AudioBook
2. Thêm màn hình: Xem chi tiết sản phẩm
3. Thay đổi yêu cầu khi load giao diện
4. Thay đổi cách tính khoảng cách, sử dụng thư viện mới
5. Thêm phương thức thanh toán mới: Thẻ nội địa (Domestic Card)
6. Thay đổi công thức tính phí vận chuyển
7. Cập nhật lại chức năng hủy đơn hàng

(5) Các hoạt động thực thi trên mã nguồn để đạt được mục tiêu kể trên

- Review mã nguồn, tìm ra những nơi có mức độ coupling, cohesion chưa hợp lý
- Tìm ra các vi phạm về SOLID trong mã nguồn
- Tìm ra các vấn đề về Clean Code trong mã nguồn
- Tìm ra những thiết kế chưa đáp ứng được tốt các yêu cầu có thể bổ sung trong tương lai
- Tái cấu trúc (refactor) lại mã nguồn để thu được mã nguồn sạch (clean code) đảm bảo tính low coupling, high cohesion, tuân thủ các nguyên tắc SOLID, ko xảy ra các vi phạm về SOLID hay coupling, cohesion ở mức kém khi thực hiện các yêu cầu có thể bổ sung trong tương lai.

(6) Kết quả dự kiến

- Thu được thông tin chi tiết về những vấn đề, vi phạm trong mã nguồn
- Giải pháp xử lý cho mỗi vấn đề, vi phạm trong mã nguồn
- Mã nguồn hoàn thiện sau khi thực hiện tái cấu trúc

1.3 Danh sách thuật ngữ

DAO: Data Access Control

SRP: Single Responsibility Principle

OCP: Open Close Principle

LSP: Liskov Substitution Principle

ISP: Interface Segregation Principle

DIP: Dependency Inversion Principle

1.4 Danh sách tham khảo

1. Centers for Medicare & Medicaid Services. (n.d.). *System Design Document Template.*

Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SystemDesignDocument.docx

2. Cornell University How We Refactor and How We Document it? On the Use of Supervised Machine Learning Algorithms to Classify Refactoring Documentation

Retrieved from www.elsevier.com/locate/eswa

2 Đánh giá thiết kế cũ

2.1 Nhận xét chung

Mã nguồn hiện tại có sử dụng các tính chất của lập trình hướng đối tượng, việc tổ chức mã nguồn có hệ thống và được phân tách thành các package rõ ràng,. Tuy nhiên, vẫn còn nhiều vấn đề trong việc thiết kế như vi phạm nguyên lý SOLID, mức độ coupling giữa các module và độ cohesion bên trong module chưa được tốt, có những đoạn code bị code smell (bad code) gây khó đọc, khó hiểu, khả năng tái sử dụng code kém, phát sinh các vấn đề như phá vỡ cấu trúc về hướng đối tượng, vi phạm các nguyên lý về hướng đối tượng khi có thêm yêu cầu mới . Nó có thể gây ra việc khó đọc hiểu mã nguồn, khó bảo trì và giảm khả năng đáp ứng các yêu cầu mới trong tương lai.

2.2 Đánh giá các mức độ coupling và cohesion

Coupling đề cập đến vấn đề phụ thuộc lẫn nhau giữa các component. Low coupling, loose coupling có nghĩa là các component ít phụ thuộc vào nhau, sự thay đổi trong component này ít khi, hoặc không ảnh hưởng đến component kia. Ngược lại, high coupling và tight coupling cho thấy các component phụ thuộc nhiều vào nhau, khi thay đổi một component thì các component kia đều bị ảnh hưởng và có khả năng phải thay đổi theo.

Cohesion thể hiện mức độ gắn kết giữa các phần tử trong một module. Các phần tử trong cùng một module phải trực tiếp hướng tới một nhiệm vụ nào đó và các phần tử cùng trực tiếp hướng tới một nhiệm vụ phải nằm trong cùng một module. High cohesion thể hiện sự gắn kết chặt chẽ giữa các phần tử trong cùng một module để thực hiện cùng một nhiệm vụ.

Trong thiết kế, ta hướng tới thiết kế low coupling và high cohesion.

2.2.1 Coupling

#	*Các mức độ về Coupling*	*Module*	*Mô tả*	*Lý do*
1	Common coupling	Controller/AuthenticationController	Các phương thức trong class này như getMainUser, login, logout đều sử dụng, chia sẻ chung global data là expired	Common coupling sẽ làm ảnh hưởng tới việc kiểm soát dữ liệu, khó xác định được tất

			Time, mainUser thuộc class sessionInformation	cả các nguồn ảnh hưởng tới dữ liệu, và khó trong việc tái sử dụng các phương thức bị common coupling
2	Common coupling	Controller/ BaseController	Các phương thức trong class này như checkMediaInCart(), getListCartMedia() đều sử dụng, chia sẻ chung global data là cartInstance thuộc class sessionInformation	Common coupling sẽ làm ảnh hưởng tới việc kiểm soát dữ liệu, khó xác định được tất cả các nguồn ảnh hưởng tới dữ liệu, và khó trong việc tái sử dụng các phương thức bị common coupling
3	Common coupling	Controller/ PlaceOrderController	Các phương thức trong class này như placeOrder(), createOrder() đều sử dụng, chia sẻ chung global data là cartInstance thuộc class sessionInformation	Common coupling sẽ làm ảnh hưởng tới việc kiểm soát dữ liệu, khó xác định được tất cả các nguồn ảnh hưởng tới dữ liệu, và khó trong việc tái sử dụng các phương thức bị common coupling
4	Stamp coupling	entity/shipping/DeliveryInfo	Phương thức CalculateShippingFee() truyền vào đối tượng order nhưng không sử dụng hết	Stamp coupling sẽ gây ảnh hưởng đến phương thức khi mà dữ liệu truyền vào thay đổi cấu trúc dữ liệu thì sẽ ảnh hưởng tới phương thức đó truy cập dữ liệu
5	Content coupling	Utils/ApplicationProgrammingInterface	Trong class này method allowMethods() có sử dụng setAccessible(true) để thay đổi sự truy cập vào data	Việc thay đổi trực tiếp đối với sự truy cập data là không an toàn

6	Stamp Coupling	Views/Screen/Home/LoginScreenHandler	Phương thức setupData truyền vào dto nhưng không sử dụng	Stamp coupling sẽ gây ảnh hưởng đến phương thức khi mà dữ liệu truyền vào thay đổi cấu trúc dữ liệu thì sẽ ảnh hưởng tới phương thức đó truy cập dữ liệu
7	Stamp Coupling	Views/Screen/IntroscreenHandler	Phương thức setupData truyền vào dto nhưng không sử dụng	Stamp coupling sẽ gây ảnh hưởng đến phương thức khi mà dữ liệu truyền vào thay đổi cấu trúc dữ liệu thì sẽ ảnh hưởng tới phương thức đó truy cập dữ liệu
8	Common Coupling	Views/Screen/Shipping/ShippingScreenHandler	Phương thức setupData sử dụng global data của lớp ShippingConfigs là PROVINCES, RUSH_SUPPORT, PROVINCES_INDEX	Common coupling sẽ làm ảnh hưởng tới việc kiểm soát dữ liệu, khó xác định được tất cả các nguồn ảnh hưởng tới dữ liệu, và khó trong việc tái sử dụng các phương thức bị common coupling
9	Stamp Coupling	entity/cart/Cart	Phương thức checkMediaInCart truyền vào cả media nhưng chỉ sử dụng mỗi thuộc tính id của media	Những thay đổi về cấu trúc dữ liệu trong media có thể làm thay đổi đến phương thức checkMediaInCart. Các dữ liệu khác về media có thể bị truy cập ngoài mong muốn

2.2.2 Cohesion

<Nhận xét tổng quát về mức độ liên kết chặt chẽ giữa các submodule>

#	*Các mức độ về Cohesion*	*Module*	*Mô tả*	*Lý do*
1	Procedural Cohesion	Controller/AuthenticationController	Phương thức logout được nhóm lại trong class này để theo tuần tự thực thi login->logout	
2	Coincidental Cohesion	Controller/AuthenticationController	Phương thức md5 nên để vào class utils để controller gọi đến khi sử dụng	Vì các phương thức không liên quan được đặt vào trong class nên dẫn tới việc quản lý và bảo trì khó khăn
3	Coincidental Cohesion	Controller/PlaceOrderController	Phương thức processDeliveryInfo không liên quan đến nghiệp vụ của class, nên tách ra một module riêng	Vì các phương thức không liên quan được đặt vào trong class nên dẫn tới việc quản lý và bảo trì khó khăn
4	Coincidental Cohesion	Controller/PlaceOrderController	Vì các phương thức validate như validateDeliveryInfo, validatePhoneNumber, validateName, validateAddress cùng xử lý logic là validate nên ta cần tách ra một module riêng để xử lý và có thể tái sử dụng được các hàm validate	Vì các phương thức không liên quan được đặt vào trong class nên dẫn tới việc quản lý và bảo trì khó khăn
5	Procedural Cohesion	entity/media/Media	Các khối lệnh trong phương thức getQuantity chỉ liên kết với nhau vì nằm trong cùng 1 trình tự thực hiện cập nhật số lượng và lấy ra số lượng	Các hành động chưa liên kết chặt chẽ với nhau, tính tái sử dụng không cao khi chỉ muốn thực hiện lấy ra số lượng mà không cập nhật
6	Communication Cohesion	entity/order/Order	Các khối lệnh trong phương thức setDeliveryInfo chỉ liên	Tính tái sử dụng chưa cao khi chỉ muốn đặt thông tin

			kết với nhau bởi dữ liệu dùng chung	vận chuyển mà không thực hiện tính phí vận chuyển
7	Coincidental Cohesion	subsystem/interbank/InterbankPayloadConverter	phương thức getToday không liên quan tới mục đích của class	Khó bảo trì, khó tái sử dụng khi phương thức này nằm ở trong một class không thể hiện mục đích của nó
8	Logical Cohesion	views/screen/popup/PopupScreen	2 phương thức success và error chỉ cùng thực hiện 1 thể loại công việc	Khó bảo trì và tái sử dụng đối với mỗi phương thức riêng lẻ

2.3 Đánh giá việc tuân theo SOLID

<Nếu phải chỉnh sửa mã nguồn theo các yêu cầu phát sinh thì bản thiết kế và mã nguồn ban đầu có tuân theo các nguyên lý thiết kế SOLID hay không?>

- Thiết kế mã nguồn hiện tại còn tồn tại một số chỗ chưa tuân theo các nguyên lý SOLID
- Khi phải chỉnh sửa mã nguồn theo các yêu cầu phát sinh sẽ xảy ra vi phạm SOLID ở một vài class

2.3.1 SRP

#	*Module*	*Mô tả*	*Lý do*
1	Controller/AuthenticationController	Class AuthenticationController thực hiện nhiều hơn một nhiệm vụ như là vừa xác thực vừa mã hóa dữ liệu	Vì các nhiệm vụ trong class này không có sự gắn kết liên quan đến nhau nên độ cohesion kém vì mong muốn chúng ta cần thiết kế mã nguồn sao cho độ cohesion trong 1 module càng cao càng tốt
2	Controller/PlaceOrderController	Class PlaceOrderController thực hiện nhiều hơn một nhiệm vụ như là vừa điều khiển luồng dữ liệu: tạo đơn	Vì có nhiều nhiệm vụ trong một class nên khi cần thay đổi hoặc phát triển thì code trong class

		hàng, tạo hóa đơn, xử lý thông tin đơn, vừa validate dữ liệu	càng ngày càng rắc rối dẫn tới khó quản lý và bảo trì hơn
3	Controller/PaymentController	Class này thực hiện nhiều hơn một nhiệm vụ vừa thực hiện validate dữ liệu date input, vừa thực hiện thanh toán qua phương thức pay Order	Khó tái sử dụng những hàm validate dữ liệu, giảm độ cohesion của module
4	Utils/ApplicationProgrammingInterface	Class thực hiện đến nhiều hơn một nhiệm vụ, vừa thực hiện get(), post() vừa tạo ra kết nối HTTP thông qua phương thức setupConnection(), đồng thời cũng sử dụng method allowMethod() cho việc điều khiển truy cập	Vì các nhiệm vụ trong class này không có sự gắn kết liên quan đến nhau nên độ cohesion kém vì mong muốn chúng ta cần thiết kế mã nguồn sao cho độ cohesion trong 1 module càng cao càng tốt đồng thời rất khó tái sử dụng code nếu sau này có các class khác mới cũng cần các phương thức connection() hay allowMethod() thì lại phải viết lại các method đó

2.3.2 OCP

#	*Module*	*Mô tả*	*Lý do*
1	Controller/PlaceOrderController	Phương thức createInvoice có tham số là một class, cụ thể là Order nếu sau này có xuất hiện loại đơn hàng khác thì sẽ ảnh hưởng đến mã nguồn	Ảnh hưởng trực tiếp đến các class như Shipping Screen Handler, deliveryInfo.. Vì sẽ phải modify để sửa code làm mất tính tái sử dụng, gây ra lỗi tiềm ẩn mà ta khó kiểm soát được sau này

2	Controller/PlaceOrderController	Phương thức validateDeliveryInfo sau này muốn thay đổi info thì phần code validate cũng cần thay đổi theo	Khi modify để sửa code thì sẽ mất tính tái sử dụng, gây ra lỗi tiềm ẩn mà ta khó kiểm soát được
3	Entity/invoice/Invoice	Phương thức Invoice có tham số là một class, cụ thể là Order nếu sau này có xuất hiện loại đơn hàng khác thì sẽ ảnh hưởng đến mã nguồn	Vì sẽ phải modify để sửa code làm mất tính tái sử dụng, gây ra lỗi tiềm ẩn mà ta khó kiểm soát được sau này
4	Entity/shipping/DeliveryInfo	Phương thức caculateShippingFee phụ thuộc vào class DistanceCalculator nên những thay đổi trong tương lai của phương thức caculateDistance sẽ làm thay đổi DeliveryInfo	Hàng loạt các class như order, PlaceOrderController … chúng ta sẽ phải modify và sửa code ở các class này khiến cho việc mất kiểm soát, test lại toàn bộ module có liên quan, đồng thời rất khó để tái sử dụng code
5	controller/PaymentController	Có trường thuộc tính *card* thuộc lớp CreditCard. Khi thêm loại thẻ mới thì code ở trong lớp này cần phải sửa theo	Giảm tính linh hoạt của mã nguồn khi đáp ứng yêu cầu mới là thêm loại thẻ mới.

2.3.3 LSP

#	*Module*	*Mô tả*	*Lý do*
1	Controller/AuthenticationController	Class AuthenticationController kế thừa BaseController nhưng lại không thực hiện các hành vi phương thức của BaseController	AuthenticationController không thể thay thế hoàn toàn BaseController, không thể thực hiện được toàn bộ các yêu cầu trong BaseController
2	Controller/PaymentController	Class PaymentController kế thừa BaseController nhưng lại	PaymentController không thể thay thế hoàn toàn BaseController, không thể

		không thực hiện các hành vi phương thức của BaseController	thực hiện được toàn bộ các yêu cầu trong BaseController
3	dao/media	Phương thức getAllMedia của lớp MediaDAO được override sai ý nghĩa tại các lớp BookDAO, CDDAO, DVDDAO.	Vì phương thức getAllMedia ở lớp MediaDAO mang ý nghĩa là lấy ra tất cả các Media từ bảng media trong database. Nhưng khi BookDAO override lại thì nó lại mang ý nghĩa là lấy tất cả các Book trong database. Hai hành vi trên bản chất là khác nhau, một cái lấy theo tiêu chí còn một cái lấy ra tất cả.

2.3.4 ISP

Mã nguồn hoàn toàn tuân thủ theo ISP

2.3.5 DIP

#	*Module*	*Mô tả*	*Lý do*
1	Controller/PlaceOrderController	Phương thức createInvoice có tham số là Order nếu sau này có xuất hiện loại đơn hàng khác thì sẽ ảnh hưởng đến mã nguồn.	Ảnh hưởng đến các class như Order, Invoice, DeliveryInfo, InvoiceScreenHandler nên khi thay đổi ở phương thức createInvoice sẽ dẫn đến các class/module phụ thuộc thayh đổi theo.
2	Entity/shipping/DeliveryInfo	Phương thức caculateShippingFree phụ thuộc vào class distanceCalculator nên những thay đổi trong tương lai của phương thức calculateDistance sẽ làm thay đổi cách tính khoảng cách ví dụ	Class DeliveryInfo đóng vai trò như là lớp low-level module cung cấp phương thức tính phí vận chuyển để các class/module như payment, Invoice, ShippingScreenHandler sử

		như thay đổi thư viện tính khoảng cách .	dụng nên khi có thay đổi trong module DeliveryInfo sẽ kéo theo các module khác thay đổi theo.
3	Controller/ PaymentControl ler	Do phương thức payOrder() ở upper-layer phụ thuộc trực tiếp vào một lớp cài đặt cụ thể là CreditCard.	Lớp Payment đóng vai trò là lớp high-level module phụ thuộc vào nhiều module ở dưới như Invoie, DeliveryInfo, PaymentScreenHandler vì vậy mà khi thay đổi ở các module ở dưới sẽ ảnh hưởng đến lớp payment.
4	dao	Các lớp trong package này thuộc persistence layer nhưng chúng không implement từ interface chung. Điều này dẫn đến việc upper-layer phải phụ thuộc trực tiếp vào các lớp DAO rời rạc.	Khi các lớp trong module này thay đổi dẫn đến việc các class ở tầng upper-layer phải sửa theo.

2.4 Các vấn đề về Clean Code

2.4.1 Clear Name

<Nhận xét tình trạng mã nguồn ban đầu có đáp ứng clear name hay không?>

#	*Module*	*Mô tả*	*Lý do*
1	views/screen/cart/ MediaHandler	Được đặt tên trùng với views/screen/home/MediaHandler và nó mang không rõ ý nghĩa của class này. Có thể chuyển thành CartMediaHandler	Giảm tính dễ hiểu của mã nguồn.

2	views/screen/popup/PopupScreen	Các phương thức static là popup(message, imgPath, undercorated), success(message), error(message), loading(message), đặt tên chưa rõ nghĩa vì không biết nó là phương thức tạo ra đối tượng thuộc lớp PopupScreen hay để show popup. Có thể chuyển thành createPopup, showSuccessPopup, showErrorPopup, showLoadingPopup	Gây ra sự không rõ ràng khi muốn sử dụng các phương thức.
	utils/MyMap	Với tên MyMap không thể hiện được nhiệm vụ của lớp này.	Về sau khi muốn sử dụng lớp này có thể phải đọc lại code của lớp này
	utils/ApplicationProgrammingInterface	Lớp này thực hiện các hành vi get(), post() vì thế tên ApplicationProgrammingInterface không mang ý nghĩa gì về các hành vi của lớp này. Tên của lớp này nên được đổi thành RequestUtils	Gây ra việc khó khăn khi cần tìm các hàm tiện ích như post(), get() về sau để sử dụng

2.4.2 Clean Function/Method

Mã nguồn hiện tại chưa đáp ứng được clean Function/Method vì một số class hiện tại có dấu hiệu code smell về Function/Method như Large loop body or deep nesting ,Data-Level Refactorings ..

#	*Module*	*Mô tả*	*Lý do*
1	Controller/AuthenticationController	Data-Level Refactorings :Trong method mainUser() có các biểu thức so sánh dài vì vậy cần chuyển vào 1 class để có thể tái sử dụng	Khó bảo trì, tái sử dụng code trong tương lai nếu một số yêu cầu cần sử dụng

2	Controller/AuthenticationController	Data-Level Refactorings : Trong method md5() có sử dụng các biến là hằng số vì vậy cần thay các biến đó bằng các tên hằng số có tính gợi nhớ, dễ đọc	Gây khó đọc hiểu,nhầm lẫn nếu người khác nhìn vào hay Một dev khác muốn sử dụng code . Sau này khi muốn thay đổi các giá trị hằng số đó sẽ phải tìm kiếm trên toàn bộ source code
3	Controller/ PlaceOrderController	Data-Level Refactorings : Trong method validateDeliveryInfo() có sử dụng các điều kiện kiểm tra về thông tin đơn hàng vì vậy có thể chuyển thành các biến kiểm tra	Khó bảo trì, tái sử dụng code trong tương lai nếu một số yêu cầu cần sử dụng
4	Controller/ PlaceOrderController	Data-Level Refactorings : Trong method validatePhoneNumber có sử dụng biến hằng số trực tiếp vì vậy cần thay bằng các tên hằng số có tính gợi nhớ, dễ đọc để có thể thay đổi được giá trị nếu cần	Sau này khi muốn thay đổi các giá trị hằng số đó sẽ phải tìm kiếm trên toàn bộ source code
5	Controller/ PlaceOrderController	Data-Level Refactorings : Cả hai method validateName() và validateAddress() đều có cùng logic xử lý nên bị duplicate code vì vậy cần viết lại thành 1 hàm và chuyển đến class utils	Bị duplicate code, khó tái sử dụng code nên cần chuyển vào class utils để các class khác có thể tái sử dụng code
6	Entity/DeliveryInfo	Data-Level Refactorings : Trong method calculateShippingFee có sử dụng hệ số làm hệ số nhân để tính phí Vì vậy	Gây khó đọc hiểu, nhầm lẫn Sau này khi muốn thay đổi các giá trị hằng số đó sẽ phải tìm kiếm trên toàn bộ source code gây mất thời gian

		cần thay đổi bằng tên hằng số có tính gợi nhớ, dễ đọc hiểu, sau này có thể thay đổi giá trị nếu cần	
7	Utils/ApplicationProgrammingInterface	Data-Level Refactorings : Trong method post() có sử dụng hằng số là mã lỗi trả về (status code) vì vậy cần viết thành 1 method riêng trong class utils cho việc trả về các status code để các class khác có thể tái sử dụng	Gây khó đọc hiểu, nhầm lẫn đối với người khác đọc code vì họ không biết đấy là các mã lỗi trả về. Khi muốn thay đổi phải tìm trên toàn bộ source code rất mất thời gian
8	App	Large loop body or deep nesting : Trong method start() làm quá nhiều việc 1 lúc vì vậy cần chia nhỏ ra thành các method	Dễ đọc, bảo trì và tái sử dụng code trong tương lai

2.4.3 Clean Class

Mã nguồn hiện tại chưa đáp ứng được clean class vì một số class hiện tại có dấu hiệu code smell về class như large class, Divergent change,..

#	*Module*	*Mô tả*	*Lý do*
1	Controller/AuthenticationController	Clean code : Large Class vì class AuthenticationController vừa làm nhiệm vụ authentication vừa phải mã hóa md5, nên tách ra 1 class utils để có thể tái sử dụng	Khó tái sử dụng code sau này nếu có thêm các class cũng cần sử dụng phần mã hóa md5 này mà muốn sử dụng lại phải viết lại hàm mã hóa tại class mới đó nên có thể gây trùng lặp đoạn code

2	Controller/ PlaceOrderController	Large Class: vì class PlaceOrderController thực hiện nhiều hơn một nhiệm vụ như vừa phải điều khiển luồng dữ liệu như tạo đơn hàng, tạo hóa đơn,xử lý thông tin đơn, vừa phải validate dữ liệu nên cần tách các phương thức validate dữ liệu ra 1 class riêng để có thể tái sử dụng ở các class khác hoặc với các yêu cầu trong tương lai.	Sẽ bị vi phạm nguyên lí SRP (Single responsibility principle) nên trong tương lai sẽ khó để bảo trì, tái sử dụng code
3	Controller/ PaymentController	Scattering (Shotgun Surgery) : vì class này đang sử dụng trực tiếp phương thức thanh toán là creditCard, trong tương lai nếu có thêm các phương thức thanh toán khác ngoài creditCard thì sẽ dẫn đến một loại các method trong các class khác nhau liên quan bị thay đổi theo.	Gây ảnh hưởng lớn vì sẽ phải modify vào nhiều class để thay đổi theo, vì vậy việc bảo trì, maintenance code rất khó đòi hỏi nhiều nỗ lực để có thể design lại hoàn toàn những thiết kế
4	entity/media	Scattering (Shotgun Surgery): vì khi ta muốn thêm 1 media mới thì sẽ phải sửa ở rất nhiều hàm liên quan đến việc load các thông số từ DB, hiển thị các kết quả ….	Gây ảnh hưởng lớn vì sẽ phải modify vào nhiều class để thay đổi theo, vì vậy việc bảo trì, maintenance code rất khó đòi hỏi nhiều nỗ lực để có thể design lại hoàn toàn những thiết kế
5	Views/CartScreenHandler,HomeScreenHandler , loginScreenHandler, ShippingScreenHandler	Alternative Classes with Different Interfaces: Duplication code do các class đều có chung 1 đoạn code setupData() và setupFunctionaly() nên bị	Gây dư thừa về mặt hành vi của 1 số class, lặp lại 1 số đoạn code trong các class

		lặp lại code ở các class screen này	

2.4.5 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.> d

3 Đề xuất cải tiến

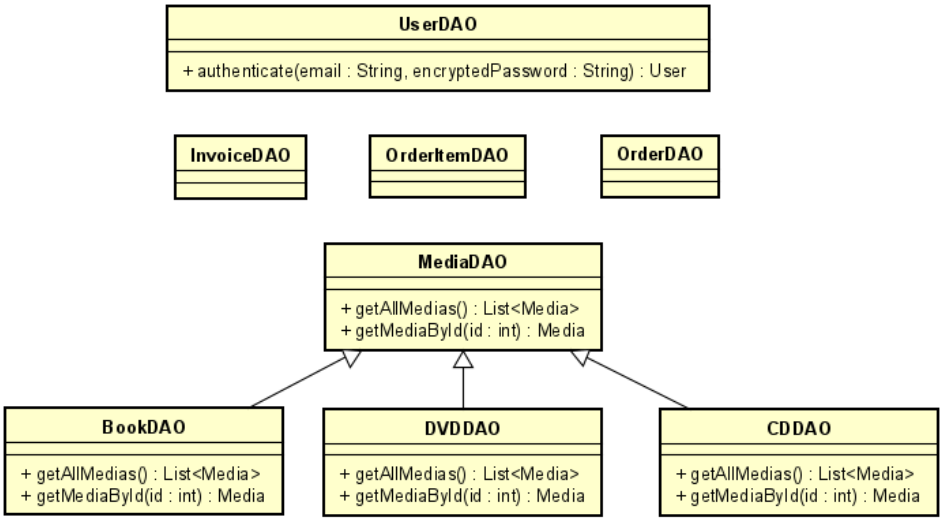
<Mục này đưa ra những đề xuất cần cải tiến để khắc phục những vấn đề trên nếu có những yêu cầu phát sinh. Lưu ý, chỉ là cải tiến cho thiết kế và mã nguồn ban đầu để trong tương lai nếu có những yêu cầu phát sinh đưa ra thì không vi phạm hoặc ít vi phạm nhất các nguyên lý thiết kế SOLID đã nêu ở trên, đồng thời đảm bảo thiết kế đạt Low Coupling và High Cohesion>

<Phần này có thể gom một số vấn đề liên quan đã nêu ở phần 2 để đưa ra giải pháp giải quyết các vấn đề này. Mỗi mục con trong phần này là 1 vấn đề đã gom lại và đưa ra giải pháp tương ứng>

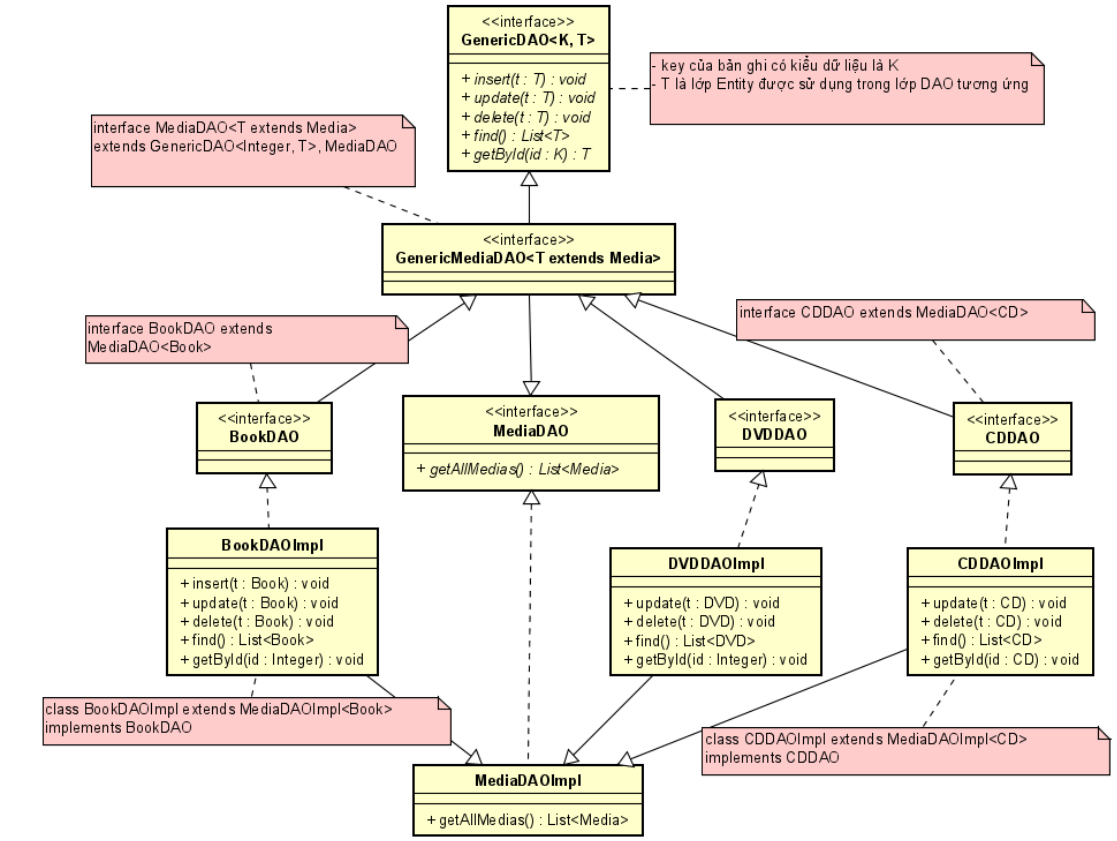
<Lưu ý, nhóm khi đưa ra đề xuất cần thể hiện các ý tưởng thiết kế dưới dạng biểu đồ lớp, biểu đồ tương tác và minh hoạ mã nguồn/ý tưởng về mã nguồn trong báo cáo; đồng thời, nhóm cần tái cấu trúc mã nguồn trên github tương ứng với giải pháp thiết kế đã đề xuất>.

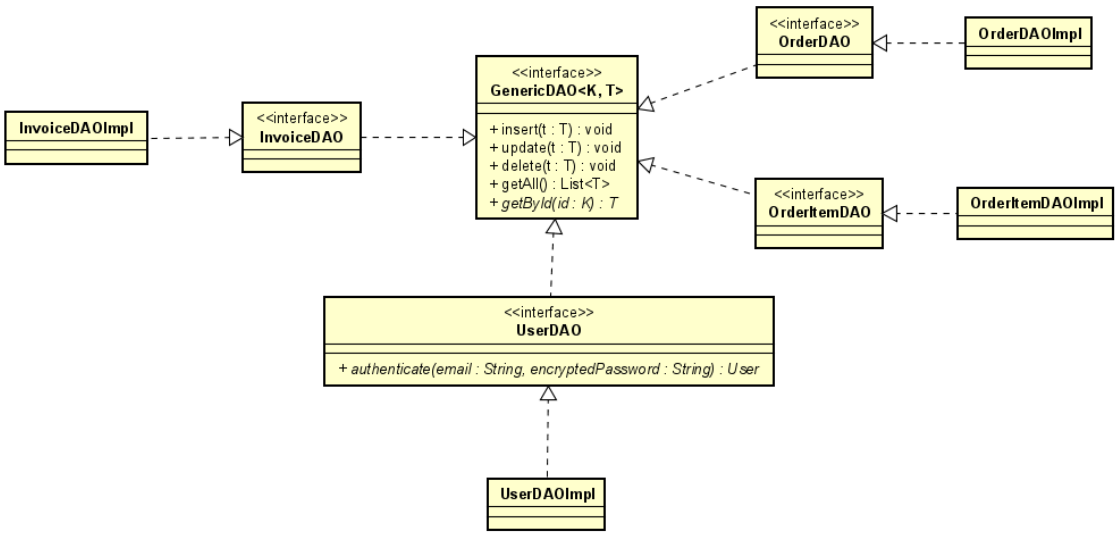
3.1 Vấn đề module DAO vi phạm nguyên lý DIP

- Trong thiết kế cũ, các lớp trong package dao thuộc persistence layer nhưng chúng không implement từ interface chung. Điều này dẫn đến việc upper-layer phải phụ thuộc trực tiếp vào các lớp DAO rời rạc. Do đó, khi các lớp trong package dao thay đổi dẫn đến việc các class ở tầng trên phải sửa theo.



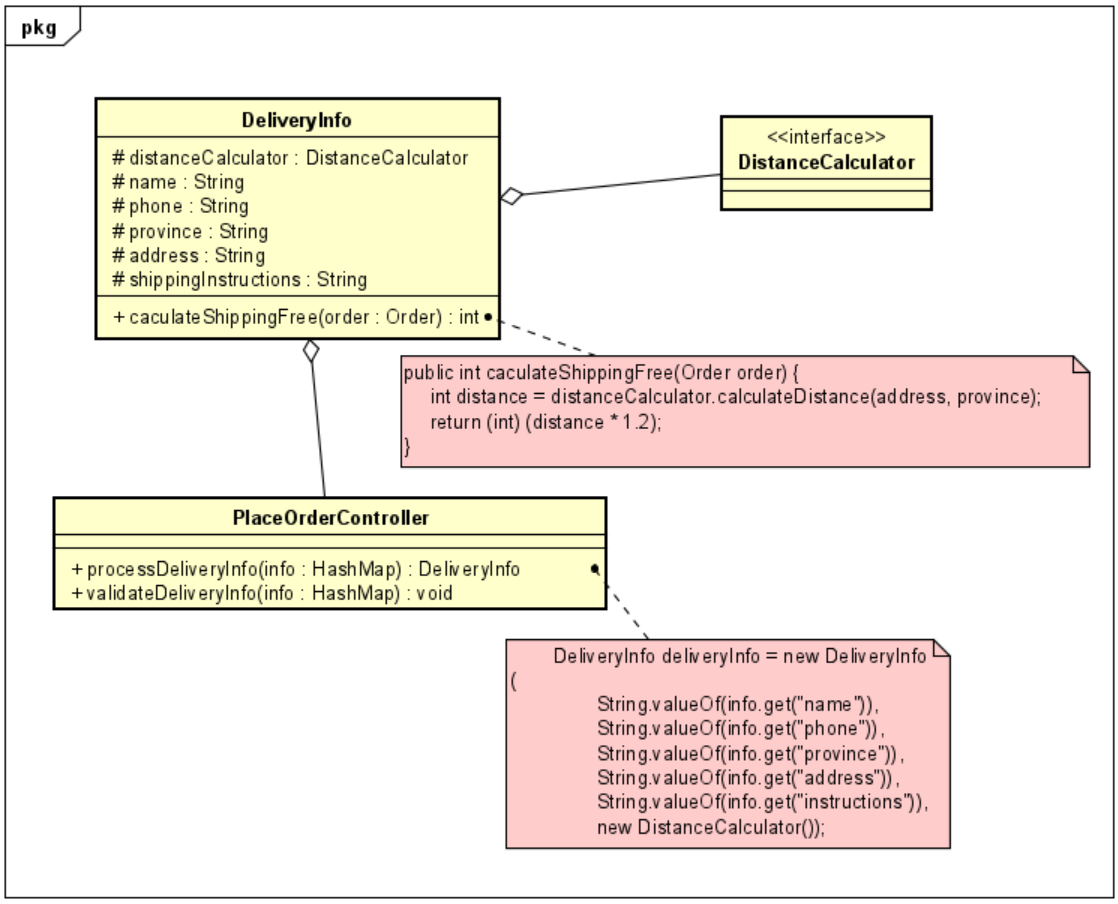
- Trong thiết kế mới này, các lớp DAO phải implements từ interface của nó. Khi ta thay đổi code bên trong các lớp DAOImpl thì các lớp ở tầng trên sẽ không bị ảnh hưởng. Hơn thế nữa, về sau khi ta muốn đổi từ SQLite sang MySQL chẳng hạn, thì ta chỉ việc viết các class implements lại các interface DAO.


<<interface>>
GenericDAO<K, T>
+ insert(t : T) : void
+ update(t : T) : void
+ delete(t : T) : void
+ find() : List<T>
+ getById(id : K) : T
- key của bản ghi có kiểu dữ liệu là K
- T là lớp Entity được sử dụng trong lớp DAO tương ứng
interface MediaDAO<T extends Media>
extends GenericDAO<Integer, T>, MediaDAO
<<interface>>
GenericMediaDAO<T extends Media>
interface BookDAO extends
MediaDAO<Book>
interface CDDAO extends MediaDAO<CD>
<<interface>>
BookDAO
<<interface>>
MediaDAO
+ getAllMedias() : List<Media>
<<interface>>
DVDDAO
<<interface>>
CDDAO
BookDAOImpl
+ insert(t : Book) : void
+ update(t : Book) : void
+ delete(t : Book) : void
+ find() : List<Book>
+ getById(id : Integer) : void
DVDDAOImpl
+ update(t : DVD) : void
+ delete(t : DVD) : void
+ find() : List<DVD>
+ getById(id : Integer) : void
CDDAOImpl
+ update(t : CD) : void
+ delete(t : CD) : void
+ find() : List<CD>
+ getById(id : CD) : void
class BookDAOImpl extends MediaDAOImpl<Book>
implements BookDAO
class CDDAOImpl extends MediaDAOImpl<CD>
implements CDDAO
MediaDAOImpl
+ getAllMedias() : List<Media>

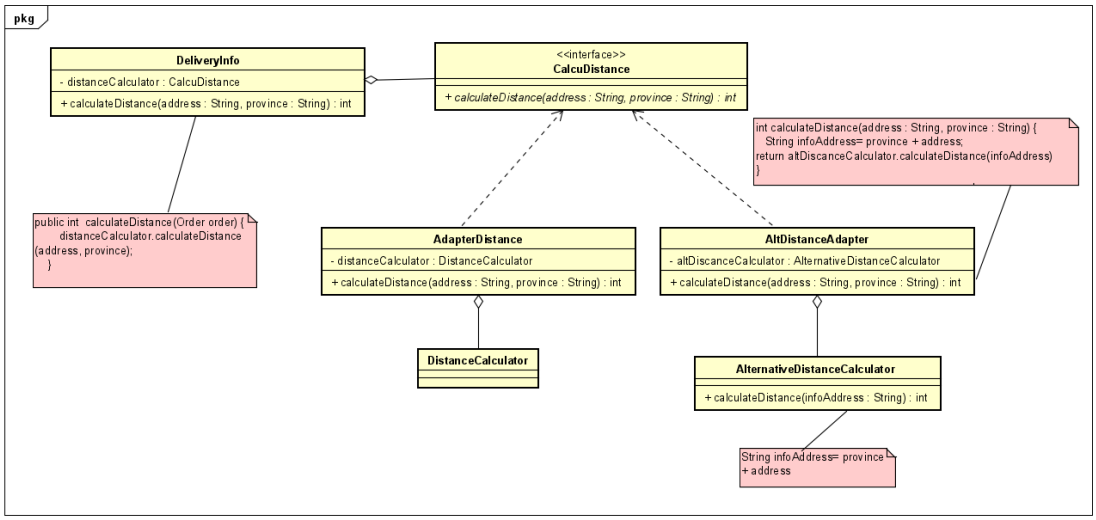

InvoiceDAOImpl
<<interface>>
InvoiceDAO
<<interface>>
GenericDAO<K, T>
+ insert(t : T) : void
+ update(t : T) : void
+ delete(t : T) : void
+ getAll() : List<T>
+ getById(id : K) : T
<<interface>>
OrderDAO
OrderDAOImpl
<<interface>>
OrderItemDAO
OrderItemDAOImpl
<<interface>>
UserDAO
+ authenticate(email : String, encryptedPassword : String) : User
UserDAOImpl

3.2 Vấn đề thay đổi thư viện tính khoảng cách (yêu cầu 4)

- Thiết kế ban đầu



- Vi phạm Nguyên Lý OCP như đã trình bày ở mục 2.3
- Lý do cải tiến :Trong tương lai khi thay đổi thư viện tính khoảng cách khác thì Delivery Info sẽ bị thay đổi, chúng ta sẽ phải chọc vào class có liên quan đến để sử đổi code vì hiện tại phương thức tính khoảng cách của thư viện cũ cũng đã được đóng gói bên trong thư viện rồi vì vậy nếu muốn sửa lại thì sẽ ảnh hưởng đến các class, module liên quan, phải test lại toàn bộ hệ thống, toàn bộ module có liên quan đến class, module chúng ta muốn chỉnh sửa làm tăng gây tiềm ẩn lỗi trong hệ thống.
- Cải tiến thiết kế : Áp dụng thiết kế Adapter Pattern

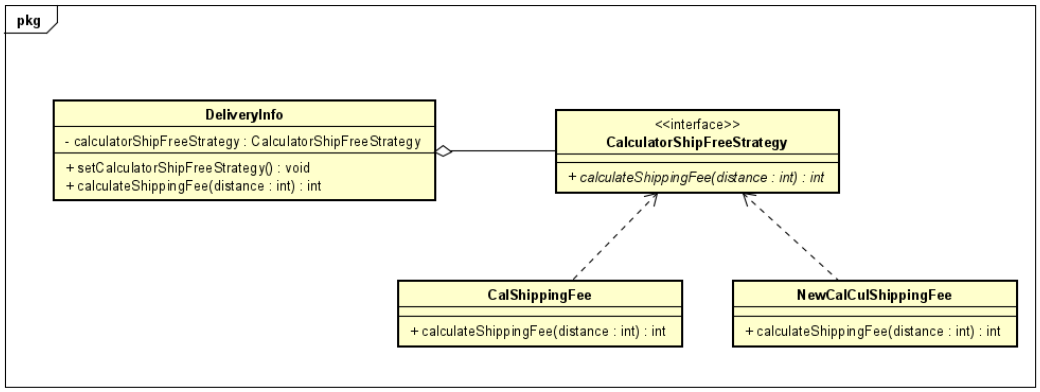


Với thiết kế trên interface calcuDistance sẽ đóng vai trò giao tiếp với application AdapterDistance và AltAdapterDistance đóng vai trò chuyển đổi (hay wrapper) 2 interface tính khoảng cách thành một interface để có thể sử dụng được giúp cho các interface không tương thích có thể hoạt động được với nhau trong cùng 1 class/interface. Với thiết kế như trên thì hầu như chúng ta không phải modify vào các code cũ vẫn đảm bảo được nguyên lý OCP (open for extension but closed for modification) cũng như đáp ứng được yêu cầu thêm thư viện tính khoảng cách mới.

3.3 Vấn đề thay đổi cách tính phí vận chuyển (yêu cầu 6)

- Vi phạm nguyên lý OCP VÀ DIP : Đã trình bày trong phần 2.3 Đánh giá việc tuân thủ theo SOLID.
- Lý do cải tiến : do khó khăn trong việc mở rộng code, bảo trì.Khi muốn thay đổi cách tính phí vận chuyển phải sửa đổi code cũ,phải modify vào các class liên quan đến cách tính phí vận chuyển. Vì vậy việc bảo trì, maintenance code rất khó đòi hỏi nhiều nỗ lực để có thể design lại hoàn toàn những thiết kế. Chúng ta chỉ cần tạo ra 1 interface, các class mới implement từ interface , thay đổi hành vi của interface ở trong class con implement làm cho class con hành xử khác đi như vậy vẫn đảm bảo được những gì đã tồn tại từ hệ thống, vẫn tăng được tính tái sử dụng, bảo trì code.

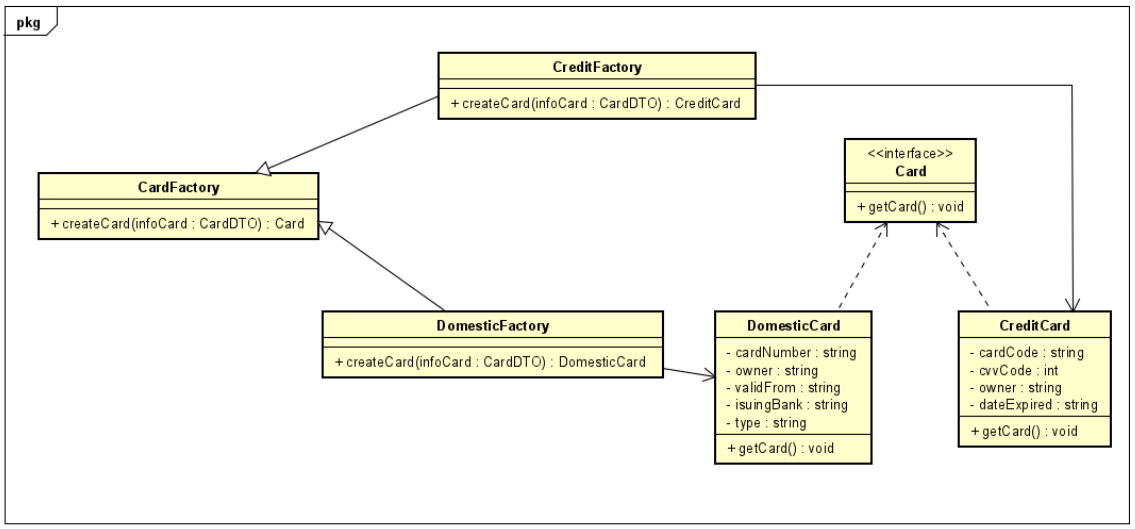
- Cải tiến thiết kế : Áp dụng mẫu thiết kế Strategy pattern



Với thiết kế trên ta thấy interface CalculateShipFeeStrategy sẽ định nghĩa các hành vi có thể có của một strategy cụ thể là cách tính phí vận chuyển còn các class calShippingFee và NewCalculShippingFee đóng vai trò như các concreteStrategy cài đặt hình vi tính phí vận chuyển cụ thể theo từng yêu cầu. DeleveriInfo sẽ chứa một tham chiếu đến đối tượng CalculateShipFeeStrategy và ủy quyền tính phí vận chuyển cho đối tượng này thực hiện. Vì vậy với thiết kế trên vẫn đảm bảo được nguyên lý OCP, chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng, thay đổi code cũ nhiều.

3.4 Vấn đề thêm phương thức thanh toán mới (Yêu cầu 5)

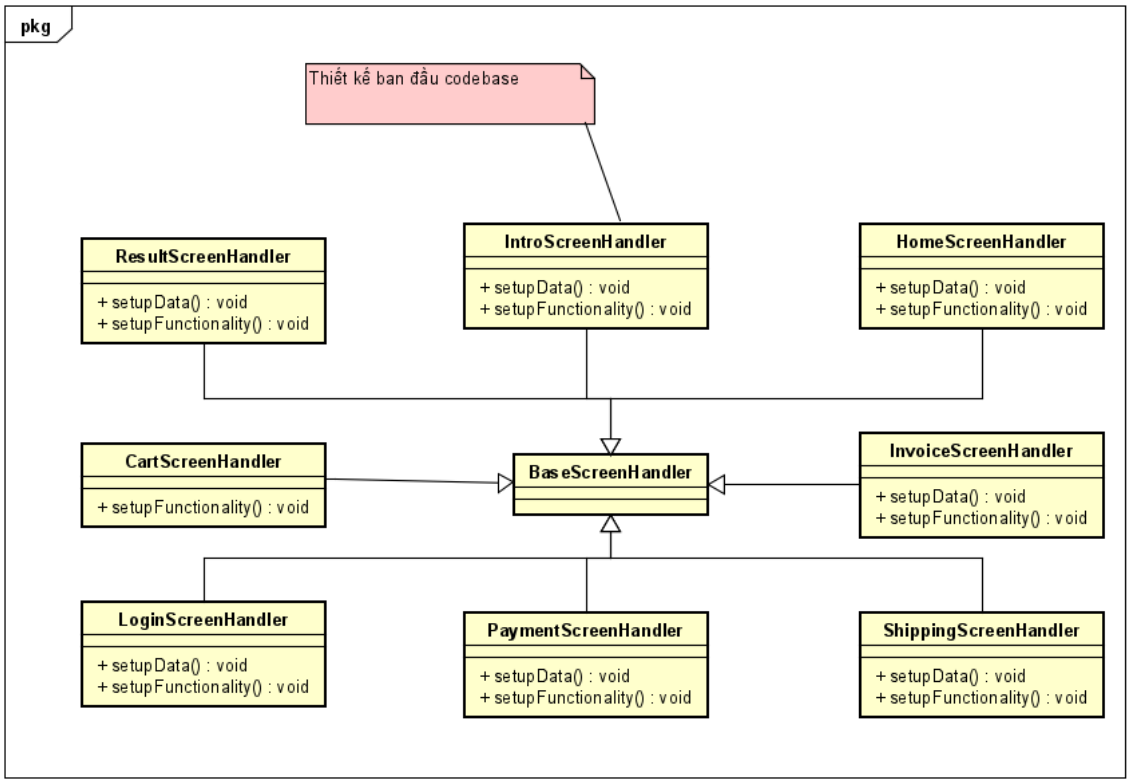
- Vi phạm nguyên lý OCP
- Lý do cải tiến :Khi thêm hình thức thanh toán mới domestic Card thì chúng ta sẽ phải modify class payment là một high-level module và các class, module khác như paymentScreenHandler, paymentTransaction ,... như vậy nếu chỉnh sửa sẽ ảnh hưởng đến các class, module liên quan, có thể sẽ phải test lại bộ module có liên quan đến class, module chúng ta muốn chỉnh sửa làm tăng gây tiềm ẩn lỗi trong hệ thống điều này vi phạm nguyên lý OCP.
- Cải tiến thiết kế : Áp dụng Factory Method



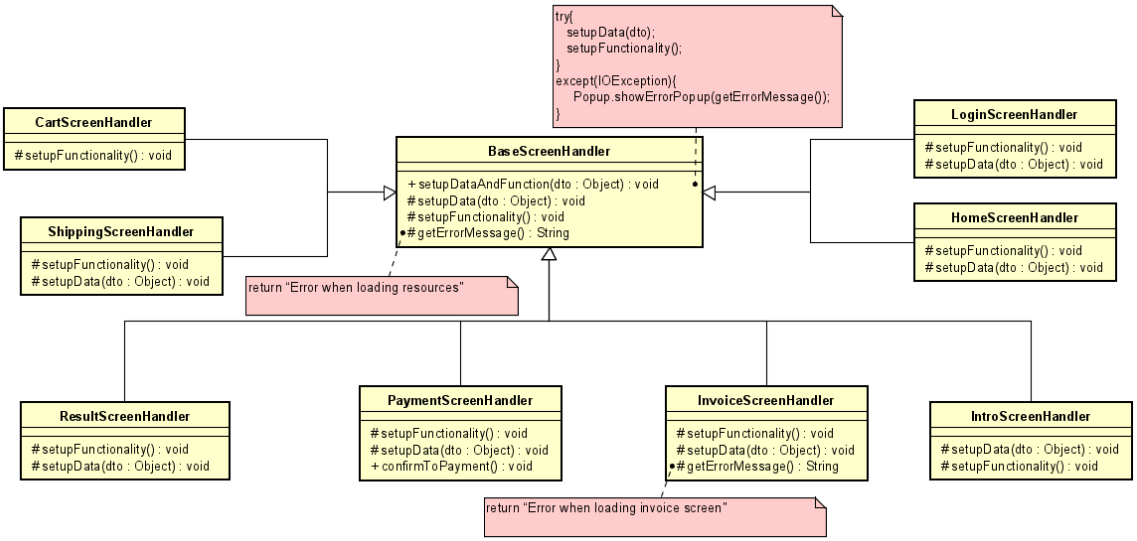
Với thiết kế trên thì card đóng vai trò là superclass để các subclass implement phương thức superclass theo cách riêng của từng subclass, card Factory đóng vai trò khởi tạo đối tượng subclass theo các tham số đầu vào vì vậy mà đã giảm được sự phụ thuộc giữa các module (loose coupling), sau này thêm 1 phương thức thanh toán thì chỉ việc tạo ra subclass và implement thêm vào superclass card

3.5 Vấn đề thay đổi yêu cầu khi load giao diện

- Thiết kế ban đầu codebase



- Lý do cải tiến: với thiết kế trước đó, các hàm setupData, setupFunctionality được gọi riêng lẻ tại các constructor lớp con của lớp BaseScreenHandler. Việc bắt lỗi IOException của các hàm setup trên cũng được bắt tại các constructor của lớp con, điều này dẫn đến sự lặp code tại nhiều class.
- Cải tiến thiết kế: sử dụng template method
- Với thiết kế mới, các phương thức setupData, setupFunctionality sẽ được gọi ở lớp cha. Lớp con chỉ làm nhiệm vụ là override các phương thức setup trên nếu cần. Với mỗi lớp con, khi cần hiển thị những lỗi riêng thì cần phải override lại phương thức getErrorMessage.

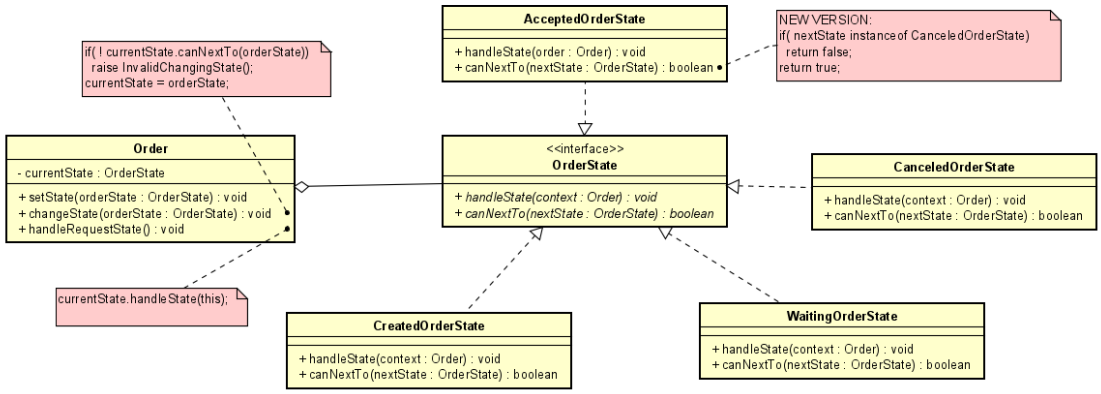


3.6 Cập nhật lại chức năng hủy đơn hàng

- Sử dụng mẫu thiết kế: State Pattern

- Với thiết kế này, sẽ đảm bảo được SRP và OCP.

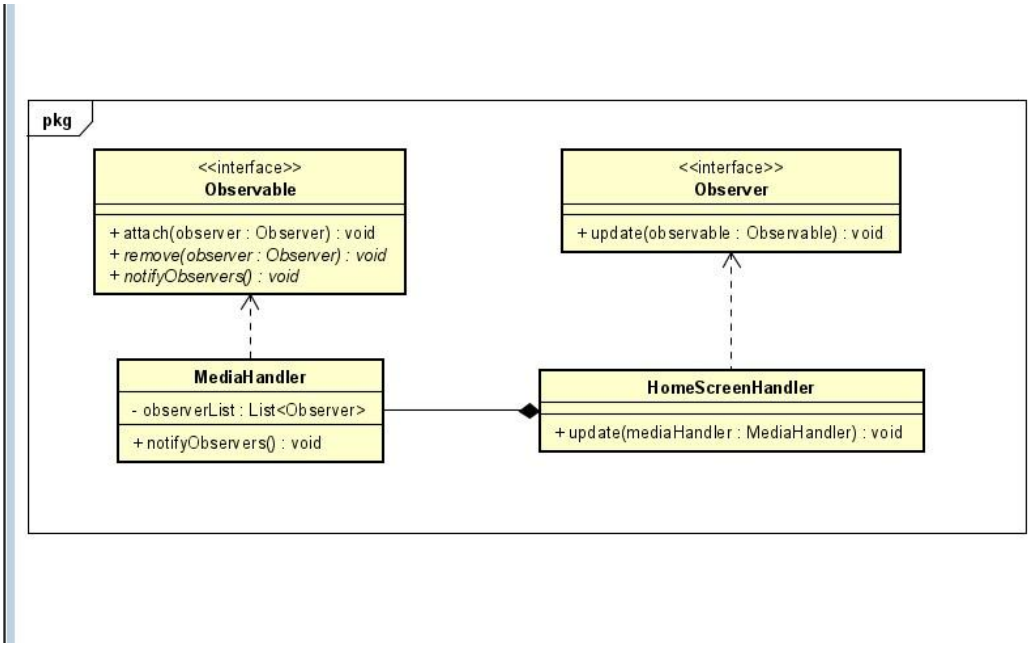
- Khi cần cập nhập phiên bản mới để đáp ứng yêu cầu "Không được hủy đơn hàng sau khi đơn hàng đã được quản trị viên phê duyệt", thì ta chỉ cần thêm điều kiện vào hàm canNextTo(Order State) của lớp Accepted Order State



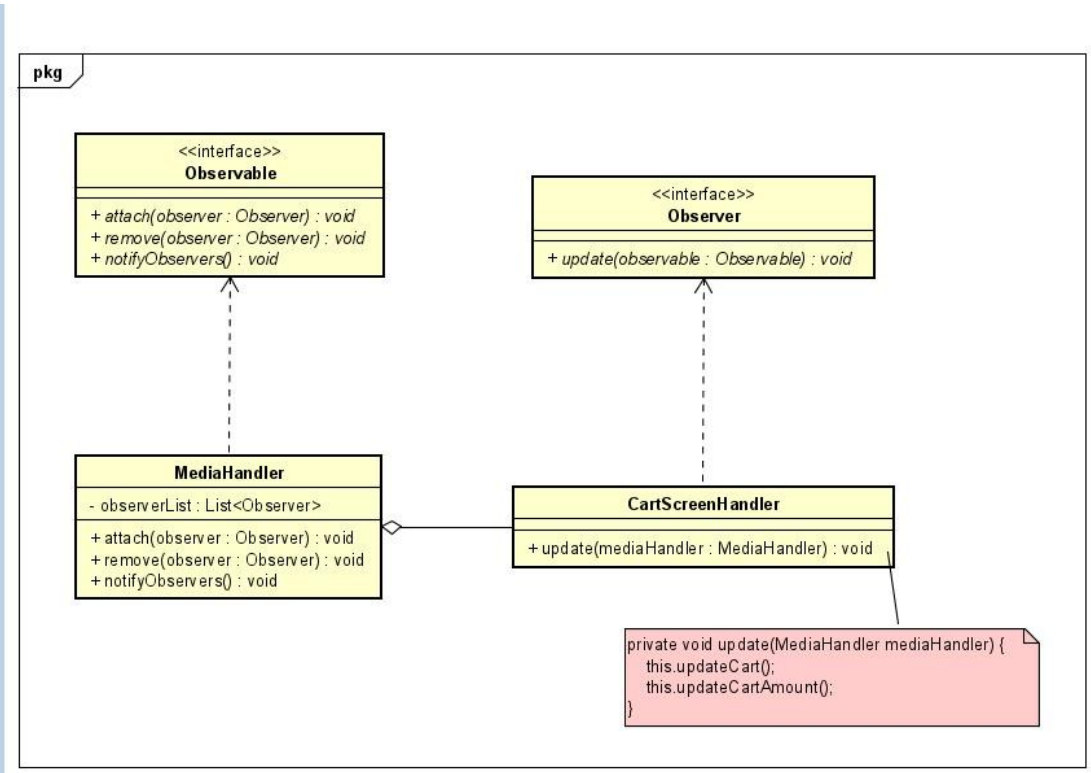
3.7 Áp dụng Observer pattern vào cart

Trong quá trình học về mẫu thiết kế Observer thì nhóm cũng đã áp dụng Observer pattern vào trong cart

Trong codebase hiện tại cũng đã áp dụng Observer pattern vào các lớp MediaHandler và HomeScreenHandler với sơ đồ mô tả bên dưới



Dựa trên cơ sở đó thì nhóm cũng đã áp dụng Observer vào cart cụ thể là các lớp CartScreenHandler và MediaHandler



Trong hình trên MediaHandler sẽ đóng vai trò là Subject để update thông tin trên các Observer đã được subscribe mỗi khi có thay đổi xảy ra còn CartScreenHandler sẽ đóng vai trò là Observer, mỗi khi nhận được thông báo thay đổi từ MediaHandler, sẽ update thông tin Cart thông qua hàm update () updateCart(), updateCartAmount().

3.8 Áp dụng Singleton Pattern vào Codebase

Trong quá trình học về mẫu thiết kế Singleton thì nhóm cũng đã áp dụng Singleton pattern vào trong cart , sessionInfomation, AIMSDB cho việc connect database nhằm đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

4 Tổng kết

4.1 Kết quả tổng quan

<So sánh với dự kiến ban đầu, kết quả đã thực hiện được hoàn thiện được bao nhiêu kỳ vọng>

So với kết quả dự kiến ban đầu thì đã tìm ra được các vấn đề vi phạm trong mã nguồn và đã đưa ra giải pháp cho các vấn đề nhưng chưa xử lý hết các vấn đề, cụ thể là phần cohesion, coupling và SOLID thì chỉ đưa ra hướng giải quyết mà chưa refactor code, chỉ refactor phần clean code. Sau khi giải quyết các vấn đề thì code đã dễ đọc dễ quản lý và dễ mở rộng hơn trong tương lai.

4.2 Các vấn đề tồn đọng

<Nêu ra các vấn đề còn sót lại trong mã nguồn. Đánh giá khả năng xử lý>

stt	*Vấn đề*	*Đánh giá khả năng xử lý*
1	Thêm mặt hàng Media mới: AudioBook	Có giải pháp xử lý nhưng không khả thi
2	Thêm màn hình: Xem chi tiết sản phẩm	Chưa có giải pháp xử lý