

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO CUỐI KỲ MÔN MẪU THIẾT KẾ

XÂY DỰNG ỨNG DỤNG GIAO ĐỒ ĂN ÁP DỤNG CÁC MẪU THIẾT KẾ

Người hướng dẫn: ThS VŨ ĐÌNH HỒNG

Người thực hiện: LÊ PHAN THẾ VĨ – 52200038

NGUYỄN CAO KỲ – 52200056

NGUYỄN NAM HOÀNG – 52200079

Lớp : 22050201

Khoa : 26

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO CUỐI KỲ MÔN MẪU THIẾT KẾ

**XÂY DỰNG ỨNG DỤNG GIAO ĐỒ ĂN
ÁP DỤNG CÁC MẪU THIẾT KẾ**

Người hướng dẫn: ThS VŨ ĐÌNH HỒNG

Người thực hiện: LÊ PHAN THẾ VĨ – 52200038

NGUYỄN CAO KỲ – 52200056

NGUYỄN NAM HOÀNG – 52200079

Lớp : 22050201

Khoa : 26

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

LỜI CẢM ƠN

Chúng em được gửi lời cảm ơn chân thành và sâu sắc nhất đến thầy Vũ Đình Hồng, người đã luôn tận tình hướng dẫn và hỗ trợ chúng em trong suốt quá trình thực hiện bài báo cáo cuối kỳ môn Mẫu thiết kế.

Trong thời gian vừa qua, thầy đã không quản ngại thời gian, luôn sẵn sàng giải đáp những thắc mắc và định hướng cho chúng em một cách kiên nhẫn, tận tâm. Nhờ sự hướng dẫn quý báu ấy, chúng em không chỉ hoàn thành bài báo cáo một cách hiệu quả và tự tin hơn, mà còn hiểu rõ hơn về nội dung môn học cũng như cách tiếp cận các vấn đề trong lĩnh vực thiết kế.

Chúng em rất mong nhận được những góp ý chân thành từ thầy để có thể tiếp tục hoàn thiện kiến thức và kỹ năng của mình trong tương lai.

Một lần nữa, chúng em xin được gửi lời cảm ơn sâu sắc đến thầy vì sự tận tâm, nhiệt huyết và hỗ trợ quý giá trong suốt thời gian qua. Chúng em hy vọng rằng sẽ còn nhiều cơ hội được học hỏi và nhận sự chỉ dẫn từ thầy trong những chặng đường sắp tới.

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của ThS Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 21 tháng 04 năm 2025

Tác giả



Lê Phan Thé Vĩ



Nguyễn Nam Hoàng



Nguyễn Cao Kỳ

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Phần đánh giá của GV chấm tai

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

TÓM TẮT

Bài báo cáo tập trung vào quá trình tái cấu trúc ứng dụng giao đồ ăn thông qua việc áp dụng các mẫu thiết kế phần mềm (design patterns). Nội dung bài sẽ trình bày tổng quan vấn đề ban đầu, giải pháp được đề xuất thông qua từng mẫu thiết kế, các trường hợp áp dụng cụ thể, kèm theo phần minh họa bằng mã nguồn để làm rõ cách triển khai trong thực tế.

MỤC LỤC

LỜI CẢM ƠN	i
PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN	iii
TÓM TẮT	iv
MỤC LỤC.....	1
DANH MỤC CÁC BẢNG BIỀU, HÌNH VẼ, ĐỒ THỊ	5
CHƯƠNG 1 – GIỚI THIỆU ỨNG DỤNG	9
1.1 Tổng quan	9
1.2 Các vấn đề về thiết kế mà ứng dụng mắc phải	9
CHƯƠNG 2 – ÁP DỤNG MẪU THIẾT KẾ	11
2.1 Singleton Pattern.....	11
2.1.1 Lý do áp dụng	11
2.1.1.1 Khái niệm Singleton Pattern	11
2.1.1.2 Lý do sử dụng	11
2.1.1.3 Các service sử dụng như thế nào?	11
2.1.2 Sơ đồ lớp	12
2.1.3 Code áp dụng	12
2.2 Template Pattern	15
2.2.1 Lý do áp dụng	15
2.2.1.1 Khái niệm Template Method	15
2.2.1.2 Trong code	15
2.2.1.3 Quy trình kết nối được định nghĩa sẵn trong connect()	15
2.2.1.4 Lý do áp dụng Template Method Pattern	15
2.2.2 Sơ đồ lớp	16
2.2.3 Code áp dụng	16
2.3 Simple Factory Pattern.....	20
2.3.1 Lý do áp dụng	20

2.3.1.1 Khái niệm.....	20
2.3.1.2 Lý do sử dụng	20
2.3.1.3 Phân tích vai trò các lớp.....	21
2.3.2 Sơ đồ lớp	21
2.3.3 Code áp dụng	21
2.4 Strategy Pattern	26
2.4.1 Lý do áp dụng	26
2.4.1.1 Khái niệm.....	26
2.4.1.2 Thành phần	26
2.4.1.3 Logic tổng thể diễn ra thế nào?.....	26
2.4.1.4 Lý do áp dụng	26
2.4.2 Sơ đồ lớp	27
2.4.3 Code áp dụng	27
2.5 Command Pattern.....	31
2.5.1 Lý do áp dụng	32
2.5.1.1 Khái niệm.....	32
2.5.1.2 Lý do áp dụng	32
2.5.1.3 Phân tích vai trò các lớp.....	33
2.5.2 Sơ đồ lớp	35
2.5.3 Code áp dụng	35
2.6 Observer Pattern.....	40
2.6.1 Lý do áp dụng	40
2.6.1.1 Khái niệm Observer Pattern.....	40
2.6.1.2 Các thành phần trong code.....	40
2.6.1.3 Lý do áp dụng Observer Pattern trong ứng dụng	40
2.6.1.4 Lợi ích cụ thể	40
2.6.1.5 Tình huống áp dụng phù hợp	41

2.6.2 Sơ đồ lớp	41
2.6.3 Code áp dụng	42
2.7 Adapter Pattern	45
2.7.1 Lý do áp dụng	45
2.7.1.1 Khái niệm.....	45
2.7.1.2 Lý do áp dụng	45
2.7.1.3 Phân tích vai trò các lớp.....	46
2.7.2 Sơ đồ lớp	47
2.7.3 Code áp dụng	47
2.8 Facade Pattern	50
2.8.1 Lý do áp dụng	50
2.8.1.1 Khái niệm Facade Pattern	50
2.8.1.2 Trong code	50
2.8.1.3 Lý do áp dụng Facade Pattern	51
2.8.2 Sơ đồ lớp	51
2.8.3 Code áp dụng	52
2.9 Proxy Pattern.....	57
2.9.1 Lý do áp dụng	57
2.9.1.1 Khái niệm.....	57
2.9.1.2 Lý do áp dụng	57
2.9.1.3 Phân tích vai trò các lớp.....	59
2.9.2 Sơ đồ lớp	61
2.9.3 Code áp dụng	61
2.10 Visitor Pattern	66
2.10.1 Lý do áp dụng	66
2.10.1.1 Khái niệm.....	66
2.10.1.2 Lý do áp dụng	66

2.10.1.3 Phân tích vai trò các lớp.....	67
2.10.2 Sơ đồ lớp	68
2.10.3 Code áp dụng	68
CHƯƠNG 3 –TỔNG KẾT.....	75

DANH MỤC KÍ HIỆU VÀ CHỮ VIẾT TẮT

CÁC KÝ HIỆU

Không có

CÁC CHỮ VIẾT TẮT

Không có

DANH MỤC CÁC BẢNG BIẾU, HÌNH VẼ, ĐỒ THỊ

DANH MỤC HÌNH CHƯƠNG 2

Hình 2. 1 Sơ đồ lớp áp dụng Singleton Pattern vào Database Connection.....	12
Hình 2. 2 Lớp DatabaseConnector	13
Hình 2. 3 Lớp ServiceCart	14
Hình 2. 4 Lớp Connection.....	14
Hình 2. 5 Sơ đồ lớp Template Method Pattern áp dụng vào Database Connector	16
Hình 2. 6 Lớp DatabaseConnector	17
Hình 2. 7 Lớp MySQLConnector	18
Hình 2. 8 Lớp PostgreSQLConnector	18
Hình 2. 9 Sơ đồ lớp áp dụng Simple Factory Pattern vào Create User Role	21
Hình 2. 10 Lớp UserRoleHandler	22
Hình 2. 11 Lớp UserRoleFactory	22
Hình 2. 12 Lớp ShipperHandler	23
Hình 2. 13 Lớp RestaurantHandler	24
Hình 2. 14 Lớp CustomerHandler.....	24
Hình 2. 15 Lớp AdminHandler	25
Hình 2. 16 Lớp ServiceUser.....	25
Hình 2. 17 Sơ đồ lớp Payment áp dụng Strategy Pattern.....	27
Hình 2. 18 Lớp PaymentContext.....	28
Hình 2. 19 Lớp PaymentStrategy	28
Hình 2. 20 Lớp BankingPayment.....	28
Hình 2. 21 Lớp CashPayment	29
Hình 2. 22 Lớp DefaultPayment	29
Hình 2. 23 Lớp ServiceCart	31
Hình 2. 24 Sơ đồ lớp áp dụng Command Pattern vào các Command của Cart	35

Hình 2. 25 Lớp ICommand	35
Hình 2. 26 Lớp CartInvoker	36
Hình 2. 27 Lớp AddMoreToCartCommand	36
Hình 2. 28 Lớp AddToCartCommand	37
Hình 2. 29 Lớp DeleteFromCartCommand	37
Hình 2. 30 Lớp MinusToCartCommand	38
Hình 2. 31 Lớp ControllerCart	39
Hình 2. 32 Sơ đồ lớp áp dụng Observer Pattern vào Order	41
Hình 2. 33 Lớp ISubject	42
Hình 2. 34 Lớp Order Manager	42
Hình 2. 35 Lớp OrderObserver	43
Hình 2. 36 Lớp CustomerOrderFood	45
Hình 2. 37 Sơ đồ lớp áp dụng Adapter Pattern vào Email Service	47
Hình 2. 38 Lớp ISsmtp	47
Hình 2. 39 Lớp IEmail	48
Hình 2. 40 Lớp SmtpAdaptee	48
Hình 2. 41 Lớp Email Adapter	49
Hình 2. 42 Lớp EmailService	50
Hình 2. 43 Sơ đồ lớp áp dụng Facade Pattern vào Voucher	52
Hình 2. 44 Lớp ServiceVoucherRestaurant	53
Hình 2. 45 Lớp ServiceVoucherSystem	55
Hình 2. 46 Lớp VoucherFacade	55
Hình 2. 47 Lớp ControllerVoucherRestaurant	56
Hình 2. 48 Lớp ControllerVoucherSystem	56
Hình 2. 49 Sơ đồ tổng quát của Proxy Pattern	59
Hình 2. 50 Sơ đồ lớp áp dụng Proxy Pattern vào Service User	61
Hình 2. 51 Lớp IServiceUser	62

Hình 2. 52 Lớp ServiceUser.....	64
Hình 2. 53 Lớp ServiceUserProxy	64
Hình 2. 54 Lớp ControllerRegister	65
Hình 2. 55 Sơ đồ lớp tổng quát Visitor Pattern.....	67
Hình 2. 56 Sơ đồ lớp áp dụng Visitor Pattern vào IUser	68
Hình 2. 57 Lớp IVisitor.....	69
Hình 2. 58 Lớp IUser	69
Hình 2. 59 Lớp Shipper.....	71
Hình 2. 60 Lớp Restaurant	71
Hình 2. 61 Lớp User.....	72
Hình 2. 62 Lớp GetAllVisitor	74
Hình 2. 63 Lớp ServiceRestaurant	74

DANH MỤC BẢNG

Bảng 2. 1 Lý do sử dụng Singleton Pattern	11
Bảng 2. 2 Thành phần của Template Method Pattern.....	15
Bảng 2. 3 Lý do áp dụng Template Method Pattern	16
Bảng 2. 4 Lý do áp dụng Simple Factory Pattern	21
Bảng 2. 5 Bảng thành phần của Strategy Pattern.....	26
Bảng 2. 6 Lý do áp dụng Strategy Pattern	27
Bảng 2. 7 Lý do áp dụng Command Pattern	33
Bảng 2. 8 Các thành phần của Command Pattern.....	33
Bảng 2. 9 Các thành phần trong Observer Pattern	40
Bảng 2. 10 Lý do áp dụng Observer Pattern	41
Bảng 2. 11 Lý do áp dụng Adapter Pattern.....	46
Bảng 2. 12 Thành phần của Adapter Pattern	46
Bảng 2. 13 Các thành phần của Facade Pattern	51

Bảng 2. 14 Lý do áp dụng Facade Pattern	51
Bảng 2. 15 Lý do áp dụng Proxy Pattern	59
Bảng 2. 16 Các thành phần của Proxy Pattern	60
Bảng 2. 17 Lý do áp dụng Visitor Pattern.....	66
Bảng 2. 18 Các thành phần của Visitor Pattern	67

CHƯƠNG 1 – GIỚI THIỆU ỨNG DỤNG

1.1 Tổng quan

Trong bối cảnh nhu cầu đặt món ăn trực tuyến ngày càng gia tăng, việc xây dựng một nền tảng kết nối hiệu quả giữa khách hàng, nhà hàng và tài xế giao hàng trở nên cần thiết hơn bao giờ hết. Ứng dụng đặt và giao món ăn ra đời nhằm giải quyết nhu cầu này, mang đến một hệ sinh thái khép kín và thuận tiện cho tất cả các bên tham gia.

Ứng dụng hoạt động như một nền tảng trung gian, giúp khách hàng dễ dàng tìm kiếm, lựa chọn món ăn yêu thích, áp dụng mã khuyến mãi, thanh toán nhanh chóng và theo dõi trạng thái đơn hàng theo thời gian thực. Đối với tài xế, ứng dụng hỗ trợ định vị, chỉ dẫn đường đi và quản lý đơn hàng một cách trực tuyến, giúp tối ưu hóa quá trình giao nhận. Nhà hàng có thể cập nhật thực đơn, xử lý đơn hàng, theo dõi doanh thu và tương tác trực tiếp với khách hàng. Trong khi đó, quản trị viên đóng vai trò giám sát toàn bộ hệ thống, quản lý người dùng, duyệt nội dung và xử lý khiếu nại phát sinh.

Hệ thống được xây dựng trên nền tảng công nghệ hiện đại, đảm bảo tính bảo mật cao và khả năng mở rộng linh hoạt, tạo nên một giải pháp toàn diện phục vụ đa đối tượng. Ứng dụng không chỉ đơn thuần là một công cụ đặt món mà còn hướng tới việc nâng cao trải nghiệm người dùng, đồng thời hỗ trợ hiệu quả cho hoạt động kinh doanh và vận hành của các nhà hàng và tài xế giao hàng.

Trong môn Công nghệ Java và Phát triển ứng dụng di động, ứng dụng đã được hoàn thành, tuy nhiên còn nhiều hạn chế - thứ mà sẽ được cải thiện trong môn Mẫu thiết kế.

1.2 Các vấn đề về thiết kế mà ứng dụng mắc phải

Mặc dù ứng dụng đã được hoàn thiện và đáp ứng đầy đủ các chức năng cần thiết, tuy nhiên trong quá trình phát triển vẫn còn tồn tại một số hạn chế về mặt thiết kế phần mềm, bao gồm:

- Không tuân thủ Coding Convention: Trong codebase hiện tại, một số đoạn mã chưa tuân theo chuẩn đặt tên biến, hàm hoặc class rõ ràng, gây khó khăn cho việc đọc hiểu và bảo trì.
- Đặt tên không nhất quán: Các class, hàm và biến được đặt tên thiếu nhất quán, đôi khi không phản ánh đúng vai trò hoặc chức năng, khiến việc tra cứu và mở rộng mã nguồn gặp nhiều trở ngại.
- Thiếu phân tách rõ ràng về trách nhiệm (Separation of Concerns): Một số thành phần trong hệ thống đảm nhiệm quá nhiều chức năng, vi phạm nguyên lý đơn trách nhiệm (Single Responsibility Principle), làm giảm tính linh hoạt khi thay đổi hoặc mở rộng.
- Khả năng mở rộng còn hạn chế: Ứng dụng hiện tại chưa được thiết kế đủ linh hoạt để dễ dàng tích hợp thêm các chức năng mới như hệ thống đánh giá, hệ thống đề xuất món ăn, hay mở rộng sang nhiều khu vực địa lý.
- Chưa áp dụng đầy đủ các Design Pattern phù hợp: Một vài phần trong hệ thống có thể tận dụng các mẫu thiết kế (như Strategy, Visitor, Adapter...) để giảm sự phụ thuộc giữa các module, tăng khả năng tái sử dụng và bảo trì, nhưng hiện chưa được áp dụng.

Những vấn đề này tuy không ảnh hưởng đến chức năng chính của hệ thống, nhưng sẽ là rào cản trong việc mở rộng, tối ưu hóa hoặc chuyển giao cho nhóm phát triển khác trong tương lai. Do đó, cần có định hướng cải thiện kiến trúc phần mềm và refactor lại mã nguồn theo các nguyên lý thiết kế hiện đại để đảm bảo tính bền vững lâu dài.

Từ các lý do trên, trong nội dung môn học này, nhóm quyết định sẽ tái cấu trúc dự án Ứng dụng giao đồ ăn nhằm cải thiện ứng dụng về mặt tổ chức, cài đặt cũng như hỗ trợ khả năng mở rộng sau này.

CHƯƠNG 2 – ÁP DỤNG MẪU THIẾT KẾ

Trong các phần bên dưới có các sơ đồ lớp, tuy nhiên trong file PDF có thể khá khó xem vì ảnh nhỏ, đây là drive ảnh các sơ đồ lớp của nhóm: <https://s.pro.vn/TIV1>.

2.1 Singleton Pattern

2.1.1 Lý do áp dụng

2.1.1.1 Khái niệm Singleton Pattern

Singleton Pattern là một **creational design pattern** nhằm đảm bảo chỉ có duy nhất một thể hiện (instance) của một lớp được tạo ra trong toàn bộ chương trình, và cung cấp điểm truy cập toàn cục (global access point) đến thể hiện đó.

2.1.1.2 Lý do sử dụng

Mục tiêu	Lý do cụ thể theo code
Tiết kiệm tài nguyên	Chỉ khởi tạo driver kết nối database một lần duy nhất.
Tránh lỗi trùng driver	Nếu nhiều Class.forName(...) được gọi đồng thời, dễ gây lỗi nạp driver.
Quản lý cấu hình tập trung	Cấu hình DB (URL, user, pass) được chia sẻ thống nhất qua một thể hiện.
Dễ mở rộng (Template + Singleton)	Kết hợp Singleton với Template Method để tạo kết nối cho nhiều loại DB (mysql, postgresql) nhưng vẫn đảm bảo 1 instance duy nhất.
Dễ bảo trì	Khi đổi config, chỉ đổi 1 lần - không phải sửa từng nơi.

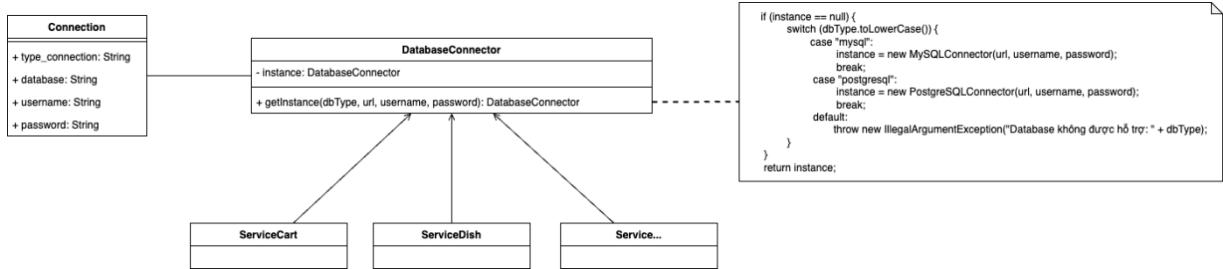
Bảng 2. 1 Lý do sử dụng Singleton Pattern

2.1.1.3 Các service sử dụng như thế nào?

Dù có bao nhiêu service như ServiceCart, ServiceVoucherRestaurant, ServiceVoucherSystem, ... sử dụng dbConnector, thì tất cả đều dùng cùng một thể hiện DatabaseConnector duy nhất.

Tóm lại: Áp dụng Singleton Pattern để đảm bảo chỉ một kết nối điều phối cơ sở dữ liệu duy nhất được sử dụng trong toàn hệ thống, giúp tiết kiệm tài nguyên, tránh trùng driver và dễ bảo trì hệ thống kết nối.

2.1.2 Sơ đồ lớp



Hình 2. 1 Sơ đồ lớp áp dụng Singleton Pattern (Simple Locking) vào Database Connection

2.1.3 Code áp dụng

```
● ● ●

public abstract class DatabaseConnector {
    protected String url;
    protected String username;
    protected String password;

    private static DatabaseConnector instance;

    protected DatabaseConnector(String url, String username, String password) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

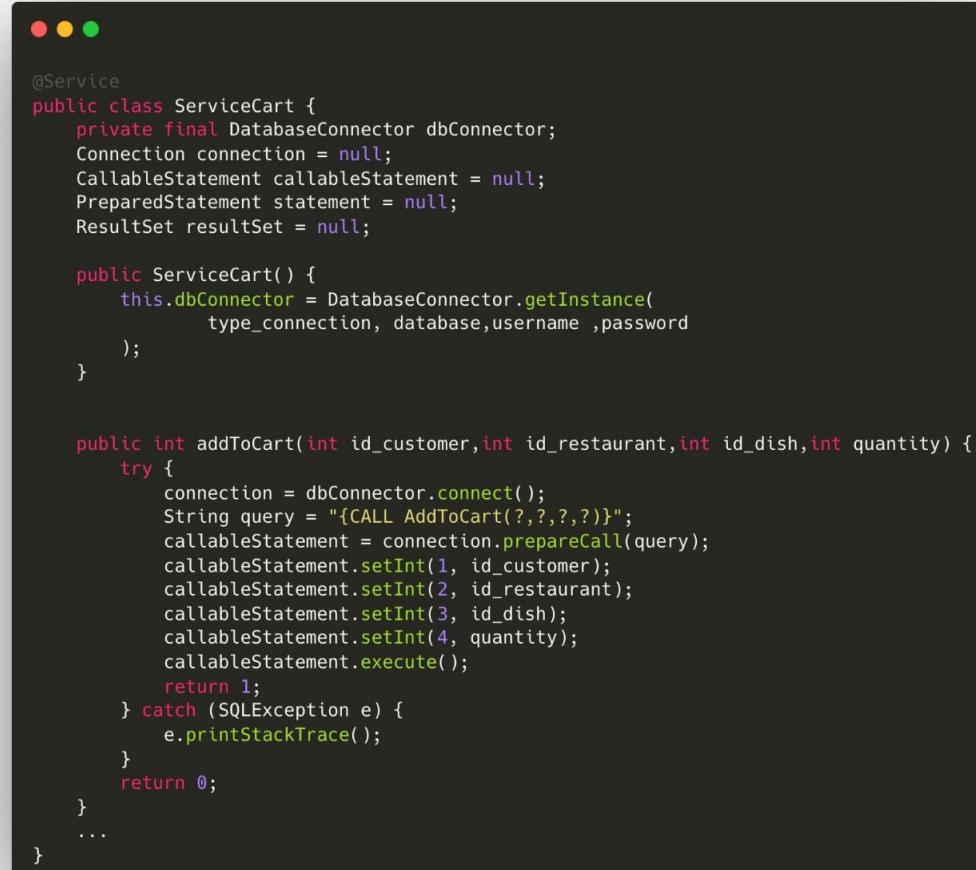
    public static synchronized DatabaseConnector getInstance(String dbType, String url, String username,
String password) {
        if (instance == null) {
            switch (dbType.toLowerCase()) {
                case "mysql":
                    instance = new MySQLConnector(url, username, password);
                    break;
                case "postgresql":
                    instance = new PostgreSQLConnector(url, username, password);
                    break;
                default:
                    throw new IllegalArgumentException("Database không được hỗ trợ: " + dbType);
            }
        }
        return instance;
    }

    public final Connection connect() {
        loadDriver();
        return createConnection();
    }

    protected abstract void loadDriver();

    private Connection createConnection() {
        try {
            return DriverManager.getConnection(url, username, password);
        } catch (SQLException e) {
            throw new RuntimeException("Kết nối database thất bại!", e);
        }
    }
}
```

Hình 2. 2 Lớp DatabaseConnector



```

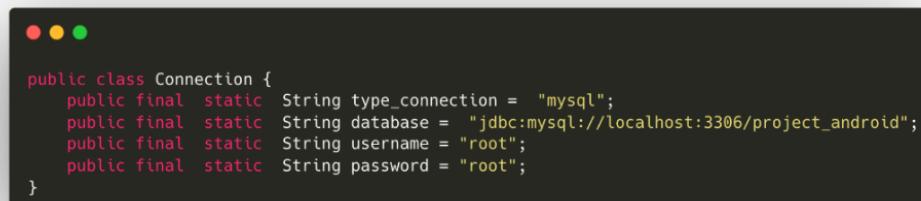
@Service
public class ServiceCart {
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    CallableStatement callableStatement = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;

    public ServiceCart() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }

    public int addToCart(int id_customer,int id_restaurant,int id_dish,int quantity) {
        try {
            connection = dbConnector.connect();
            String query = "{CALL AddToCart(?, ?, ?, ?)}";
            callableStatement = connection.prepareCall(query);
            callableStatement.setInt(1, id_customer);
            callableStatement.setInt(2, id_restaurant);
            callableStatement.setInt(3, id_dish);
            callableStatement.setInt(4, quantity);
            callableStatement.execute();
            return 1;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }
    ...
}

```

Hình 2. 3 Lớp ServiceCart



```

public class Connection {
    public final static String type_connection = "mysql";
    public final static String database = "jdbc:mysql://localhost:3306/project_android";
    public final static String username = "root";
    public final static String password = "root";
}

```

Hình 2. 4 Lớp Connection - cấu hình loại CSDL

2.2 Template Pattern

2.2.1 Lý do áp dụng

2.2.1.1 Khái niệm Template Method

Template Method Pattern là một **behavioral design pattern** (mẫu thiết kế hành vi) cho phép: ta xác định khung (template) của một thuật toán trong một phương thức, và để các lớp con định nghĩa các bước cụ thể mà không thay đổi cấu trúc của thuật toán đó.

2.2.1.2 Trong code

Thành phần	Mã nguồn cụ thể
Lớp trừu tượng (abstract class)	DatabaseConnector
Template method	connect()
Các bước được cố định	createConnection()
Các bước có thể thay đổi (abstract)	loadDriver()
Lớp con định nghĩa chi tiết	MySQLConnector, PostgreSQLConnector
Các lớp sử dụng	Các service: ServiceCart, ServiceDish,...

Bảng 2. 2 Thành phần của Template Method Pattern

2.2.1.3 Quy trình kết nối được định nghĩa sẵn trong connect()

- loadDriver() là **hook method** – được lớp con định nghĩa tùy biến.
- createConnection() là phần **dùng chung cho tất cả DB**, không cần thay đổi.

2.2.1.4 Lý do áp dụng Template Method Pattern

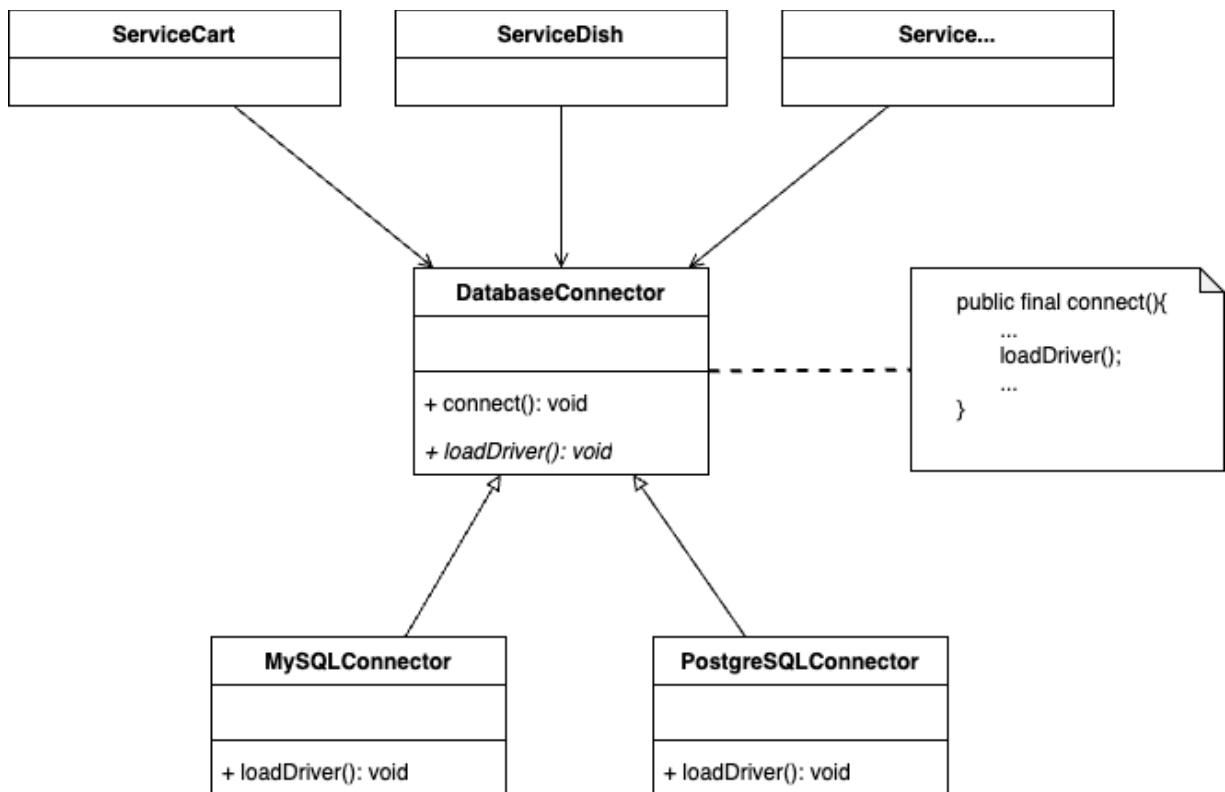
Lý do	Mô tả cụ thể trong code
Tái sử dụng quy trình cố định	Tất cả các loại DB đều dùng chung connect() với quy trình 2 bước.
Tách riêng phần thay đổi	Chỉ loadDriver() khác nhau → lớp con định nghĩa lại.
Dễ mở rộng	Thêm loại DB mới chỉ cần tạo lớp con mới, không cần đụng vào code cũ. Ví dụ với Oracle: chỉ cần tạo OracleConnector và override loadDriver()

Giảm trùng lắp	Không phải viết lại toàn bộ quy trình kết nối ở từng loại DB.
Dảm bảo quy trình chuẩn	Luôn luôn gọi đúng thứ tự: load driver → connect. Không bị lỗi do lặp sai.

Bảng 2. 3 Lý do áp dụng Template Method Pattern

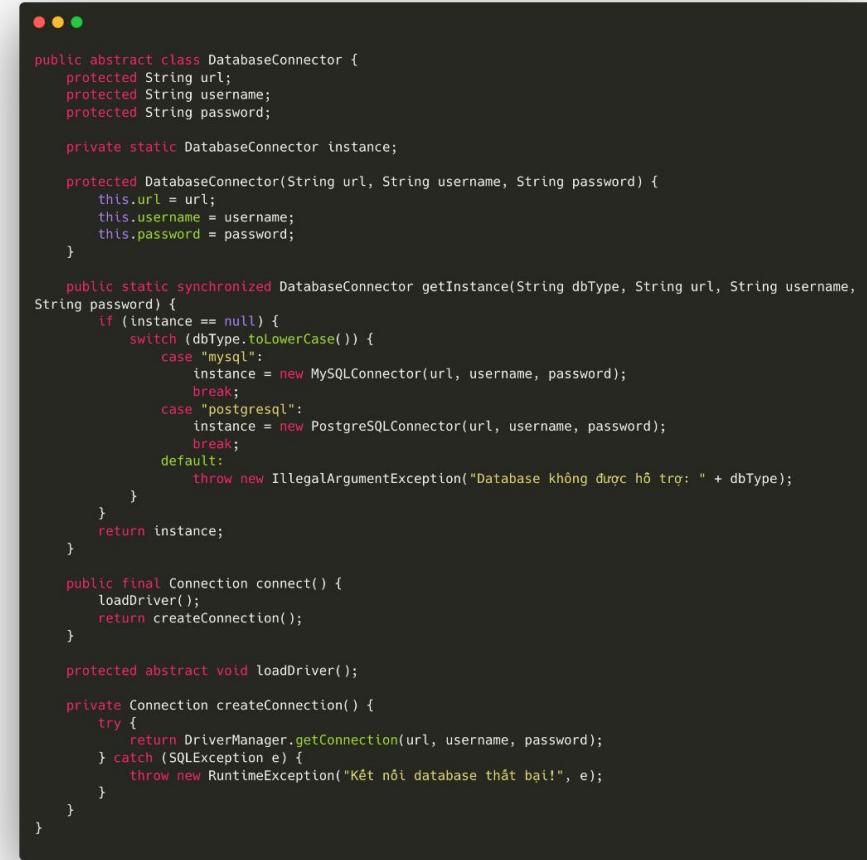
Tóm lại: Áp dụng Template Method Pattern để định nghĩa quy trình kết nối cơ sở dữ liệu trong connect() một cách chuẩn hóa, đồng thời cho phép các lớp con như MySQLConnector và PostgreSQLConnector,... tùy biến phần nạp driver mà không làm thay đổi cấu trúc tổng thể.

2.2.2 Sơ đồ lớp



Hình 2. 5 Sơ đồ lớp Template Method Pattern áp dụng vào Database Connector

2.2.3 Code áp dụng



```
public abstract class DatabaseConnector {
    protected String url;
    protected String username;
    protected String password;

    private static DatabaseConnector instance;

    protected DatabaseConnector(String url, String username, String password) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

    public static synchronized DatabaseConnector getInstance(String dbType, String url, String username,
String password) {
        if (instance == null) {
            switch (dbType.toLowerCase()) {
                case "mysql":
                    instance = new MySQLConnector(url, username, password);
                    break;
                case "postgresql":
                    instance = new PostgreSQLConnector(url, username, password);
                    break;
                default:
                    throw new IllegalArgumentException("Database không được hỗ trợ: " + dbType);
            }
        }
        return instance;
    }

    public final Connection connect() {
        loadDriver();
        return createConnection();
    }

    protected abstract void loadDriver();

    private Connection createConnection() {
        try {
            return DriverManager.getConnection(url, username, password);
        } catch (SQLException e) {
            throw new RuntimeException("Kết nối database thất bại!", e);
        }
    }
}
```

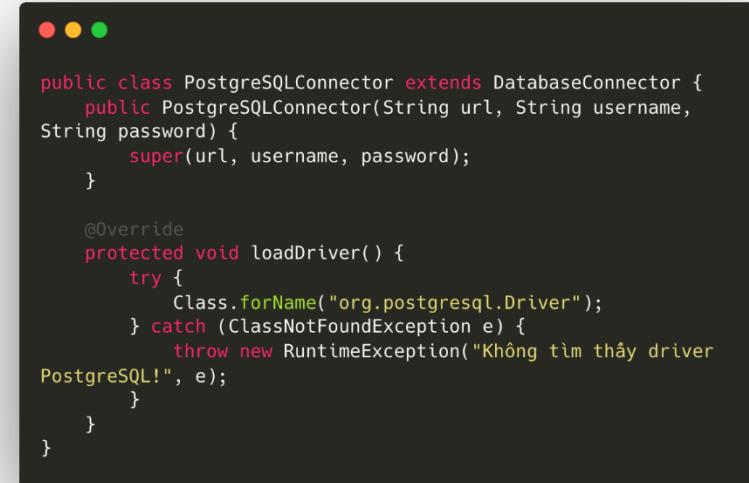
Hình 2. 6 Lớp DatabaseConnector



```
public class MySQLConnector extends DatabaseConnector {
    public MySQLConnector(String url, String username, String password) {
        super(url, username, password);
    }

    @Override
    protected void loadDriver() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Không tìm thấy driver MySQL!", e);
        }
    }
}
```

Hình 2. 7 Lớp MySQLConnector



```
public class PostgreSQLConnector extends DatabaseConnector {
    public PostgreSQLConnector(String url, String username,
String password) {
        super(url, username, password);
    }

    @Override
    protected void loadDriver() {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Không tìm thấy driver
PostgreSQL!", e);
        }
    }
}
```

Hình 2. 8 Lớp PostgreSQLConnector

```

@Service
public class ServiceCart {
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    CallableStatement callableStatement = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;

    public ServiceCart() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }

    public int addToCart(int id_customer,int id_restaurant,int id_dish,int quantity) {
        try {
            // Sử dụng tại đây
            connection = dbConnector.connect();
            String query = "{CALL AddToCart(?, ?, ?, ?)}";
            callableStatement = connection.prepareCall(query);
            callableStatement.setInt(1, id_customer);
            callableStatement.setInt(2, id_restaurant);
            callableStatement.setInt(3, id_dish);
            callableStatement.setInt(4, quantity);
            callableStatement.execute();
            return 1;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }

    public int addMoreToCart(String id_order,int id_dish) {
        try {
            // Sử dụng tại đây
            connection = dbConnector.connect();
            String query = "{CALL AddMoreToCart(?, ?)}";
            callableStatement = connection.prepareCall(query);
            callableStatement.setString(1, id_order);
            callableStatement.setInt(2, id_dish);
            callableStatement.execute();
            return 1;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }
    ...
}

```

Hình 2. 9 Lớp ServiceCart

2.3 Simple Factory Pattern

2.3.1 Lý do áp dụng

2.3.1.1 Khái niệm

Factory Pattern là một mẫu thiết kế thuộc nhóm Creational Design Patterns, được sử dụng để tạo đối tượng mà không cần phải chỉ định chính xác lớp đối tượng cần được khởi tạo. Mẫu này giúp tách biệt quá trình tạo đối tượng khỏi mã sử dụng đối tượng đó, cho phép dễ dàng mở rộng và bảo trì khi cần thay đổi cách thức tạo đối tượng.

Simple Factory (hay còn gọi là Factory Method đơn giản) là một biến thể của Factory Pattern, trong đó một lớp duy nhất chịu trách nhiệm tạo các đối tượng khác nhau, mà không yêu cầu các lớp con phải quyết định cách thức tạo đối tượng. Lớp Factory có thể chứa một phương thức duy nhất để khởi tạo các đối tượng dựa trên thông tin đầu vào.

2.3.1.2 Lý do sử dụng

Mục tiêu	Lý do áp dụng Simple Factory
Tiết kiệm tài nguyên	Simple Factory giúp tạo ra các đối tượng cần thiết mà không cần phải khởi tạo trực tiếp từng lớp con. Điều này giúp tối ưu tài nguyên và tránh lặp lại logic tạo đối tượng.
Quản lý cấu hình tập trung	Cấu hình khởi tạo đối tượng (như database connection) được quản lý qua Factory. Việc thay đổi cấu hình chỉ cần thực hiện tại Factory mà không cần thay đổi ở các lớp khác.
Dễ mở rộng	Khi thêm mới loại Handler (ví dụ: ManagerHandler), chỉ cần cập nhật trong Factory mà không cần thay đổi mã nguồn ở các phần khác của hệ thống.

DỄ BẢO TRÌ	Việc khởi tạo đối tượng được tập trung trong Factory, giúp dễ dàng bảo trì và thay đổi cách tạo đối tượng mà không ảnh hưởng đến các phần khác trong ứng dụng.
------------	--

Bảng 2. 4 Lý do áp dụng Simple Factory Pattern

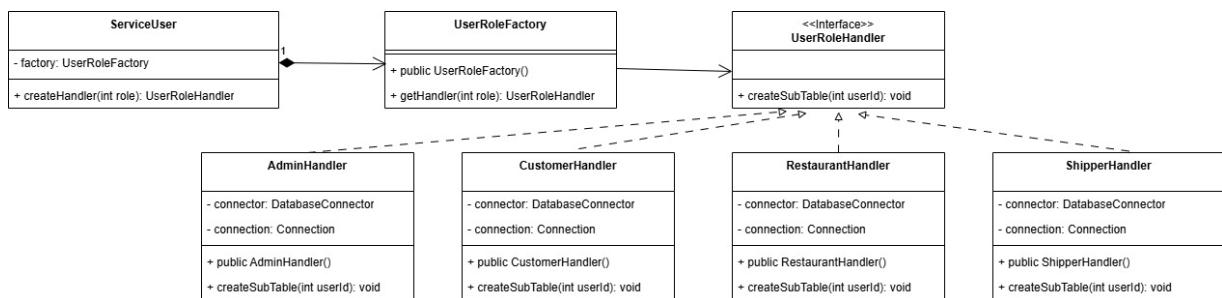
2.3.1.3 Phân tích vai trò các lớp

Dùng UserRoleFactory để tạo các đối tượng UserRoleHandler (như AdminHandler, CustomerHandler) dựa trên vai trò người dùng, thay vì khởi tạo từng đối tượng trực tiếp ở các lớp khác.

Mọi đối tượng Handler được khởi tạo với cùng một cấu hình (kết nối database), giúp tiết kiệm thời gian và tránh lặp lại mã nguồn.

Khi cần thêm loại Handler mới (ví dụ: ManagerHandler), chỉ cần cập nhật Factory mà không phải thay đổi các lớp khác. Tất cả logic khởi tạo đối tượng được quản lý ở một nơi duy nhất, giúp dễ dàng bảo trì và thay đổi sau này.

2.3.2 Sơ đồ lớp



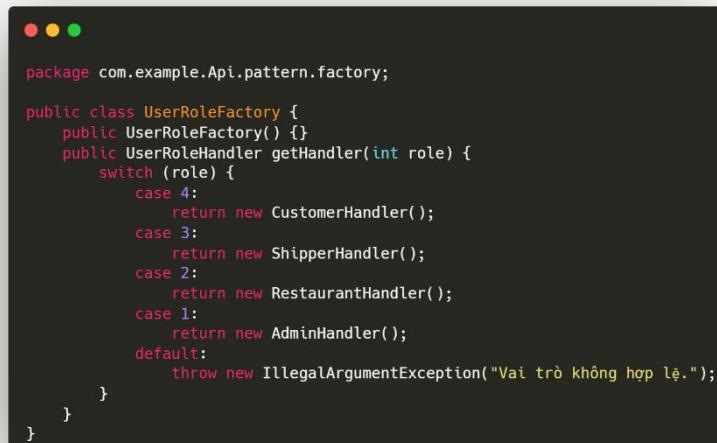
Hình 2. 10 Sơ đồ lớp áp dụng Simple Factory Pattern vào Create User Role

2.3.3 Code áp dụng



```
package com.example.Api.pattern.factory;
public interface UserRoleHandler {
    void createSubTable(int userId);
}
```

Hình 2. 11 Lớp UserRoleHandler



```
package com.example.Api.pattern.factory;
public class UserRoleFactory {
    public UserRoleFactory() {}
    public UserRoleHandler getHandler(int role) {
        switch (role) {
            case 4:
                return new CustomerHandler();
            case 3:
                return new ShipperHandler();
            case 2:
                return new RestaurantHandler();
            case 1:
                return new AdminHandler();
            default:
                throw new IllegalArgumentException("Vai trò không hợp lệ.");
        }
    }
}
```

Hình 2. 12 Lớp UserRoleFactory



```
package com.example.Api.pattern.factory;

import com.example.Api.pattern.template.DatabaseConnector;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;

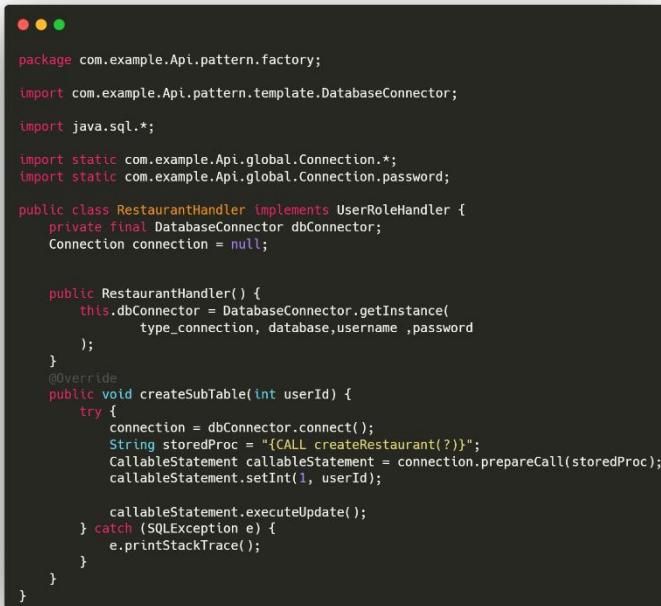
import static com.example.Api.global.Connection.*;
import static com.example.Api.global.Connection.password;

public class ShipperHandler implements UserRoleHandler {
    private final DatabaseConnector dbConnector;
    Connection connection = null;

    public ShipperHandler() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }
    @Override
    public void createSubTable(int userId) {
        try {
            Connection connection = dbConnector.connect();
            String storedProc = "{CALL createShiper(?)}";
            CallableStatement callableStatement = connection.prepareCall(storedProc);
            callableStatement.setInt(1, userId);

            callableStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Hình 2. 13 Lớp ShipperHandler



```

package com.example.Api.pattern.factory;

import com.example.Api.pattern.template.DatabaseConnector;
import java.sql.*;

import static com.example.Api.global.Connection.*;
import static com.example.Api.global.Connection.password;

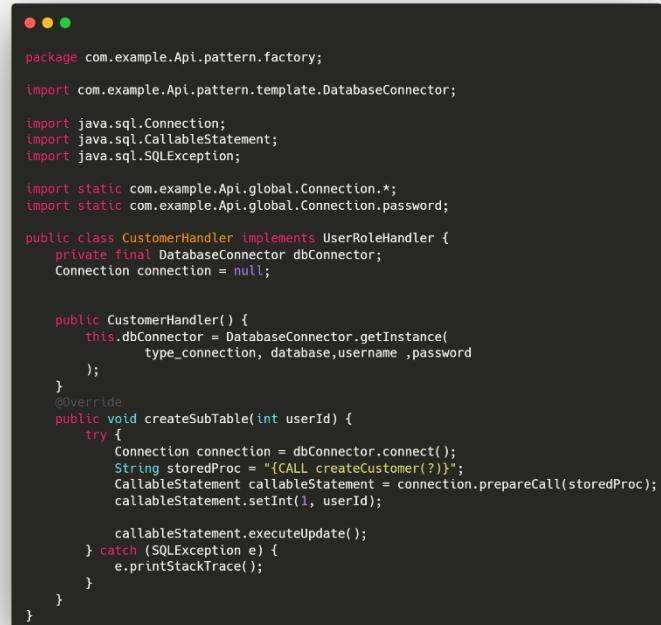
public class RestaurantHandler implements UserRoleHandler {
    private final DatabaseConnector dbConnector;
    Connection connection = null;

    public RestaurantHandler() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }
    @Override
    public void createSubTable(int userId) {
        try {
            connection = dbConnector.connect();
            String storedProc = "{CALL createRestaurant(?)}";
            CallableStatement callableStatement = connection.prepareCall(storedProc);
            callableStatement.setInt(1, userId);

            callableStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Hình 2. 14 Lớp RestaurantHandler



```

package com.example.Api.pattern.factory;

import com.example.Api.pattern.template.DatabaseConnector;
import java.sql.Connection;
import java.sql.CallableStatement;
import java.sql.SQLException;

import static com.example.Api.global.Connection.*;
import static com.example.Api.global.Connection.password;

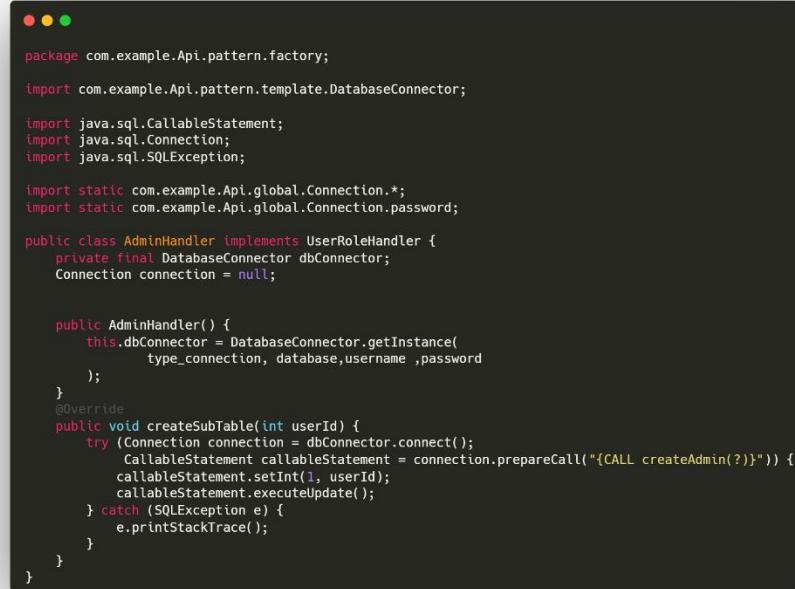
public class CustomerHandler implements UserRoleHandler {
    private final DatabaseConnector dbConnector;
    Connection connection = null;

    public CustomerHandler() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }
    @Override
    public void createSubTable(int userId) {
        try {
            Connection connection = dbConnector.connect();
            String storedProc = "{CALL createCustomer(?)}";
            CallableStatement callableStatement = connection.prepareCall(storedProc);
            callableStatement.setInt(1, userId);

            callableStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Hình 2. 15 Lớp CustomerHandler



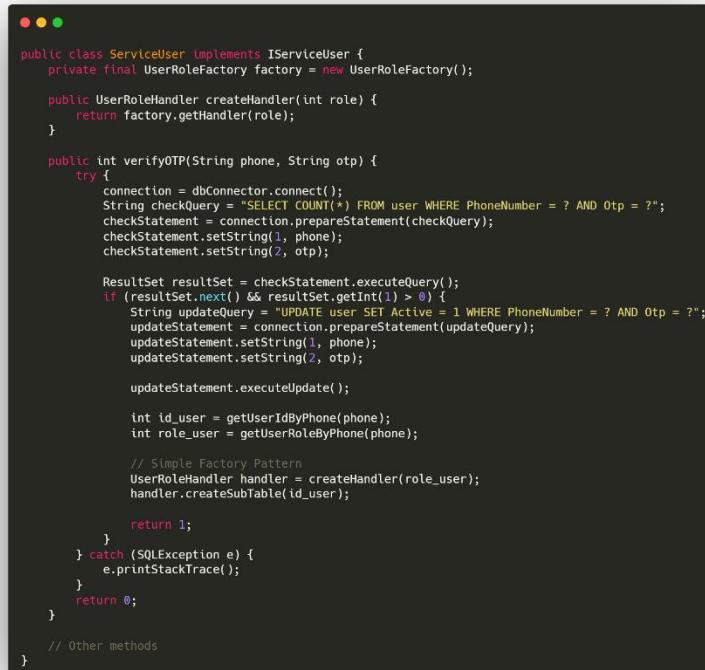
```

package com.example.Api.pattern.factory;
import com.example.Api.pattern.template.DatabaseConnector;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import static com.example.Api.global.Connection.*;
import static com.example.Api.global.Connection.password;
public class AdminHandler implements UserRoleHandler {
    private final DatabaseConnector dbConnector;
    Connection connection = null;

    public AdminHandler() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database,username ,password
        );
    }
    @Override
    public void createSubTable(int userId) {
        try (Connection connection = dbConnector.connect();
            CallableStatement callableStatement = connection.prepareCall("{CALL createAdmin(?)}")) {
            callableStatement.setInt(1, userId);
            callableStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Hình 2. 16 Lớp AdminHandler



```

public class ServiceUser implements IServiceUser {
    private final UserRoleFactory factory = new UserRoleFactory();

    public UserRoleHandler createHandler(int role) {
        return factory.getHandler(role);
    }

    public int verifyOTP(String phone, String otp) {
        try {
            connection = dbConnector.connect();
            String checkQuery = "SELECT COUNT(*) FROM user WHERE PhoneNumber = ? AND Otp = ?";
            checkStatement = connection.prepareStatement(checkQuery);
            checkStatement.setString(1, phone);
            checkStatement.setString(2, otp);

            ResultSet resultSet = checkStatement.executeQuery();
            if (resultSet.next() && resultSet.getInt(1) > 0) {
                String updateQuery = "UPDATE user SET Active = 1 WHERE PhoneNumber = ? AND Otp = ?";
                updateStatement = connection.prepareStatement(updateQuery);
                updateStatement.setString(1, phone);
                updateStatement.setString(2, otp);

                updateStatement.executeUpdate();

                int id_user = getUserIdByPhone(phone);
                int role_user = getUserRoleByPhone(phone);

                // Simple Factory Pattern
                UserRoleHandler handler = createHandler(role_user);
                handler.createSubTable(id_user);

                return 1;
            } catch (SQLException e) {
                e.printStackTrace();
            }
            return 0;
        }
        // Other methods
    }
}

```

Hình 2. 17 Lớp ServiceUser

2.4 Strategy Pattern

2.4.1 Lý do áp dụng

2.4.1.1 Khái niệm

Strategy Pattern là mẫu thiết kế hành vi (behavioral pattern) cho phép bạn **định nghĩa nhiều thuật toán khác nhau**, đóng gói chúng thành các class riêng biệt, và cho phép thay đổi thuật toán được sử dụng **mà không thay đổi code của client**.

2.4.1.2 Thành phần

Thành phần	Mô tả	Lớp cụ thể
Strategy interface	Định nghĩa chung cho các chiến lược	PaymentStrategy
Concrete strategies	Các thuật toán cụ thể (tính phí)	CashPayment, BankingPayment, DefaultPayment
Context	Sử dụng strategy và cho phép thay đổi nó	PaymentContext
Client	Gọi và sử dụng Context để xử lý logic	ServiceCart

Bảng 2. 5 Bảng thành phần của Strategy Pattern

2.4.1.3 Logic tổng thể diễn ra thế nào?

- Tạo context thanh toán: PaymentContext context = new PaymentContext(...)
- Gán chiến lược cụ thể: context.setStrategy(...)
- Tính toán phí thanh toán: paymentContext.executeStrategy(total)
- Ghi thông tin vào cơ sở dữ liệu (method + phí)

2.4.1.4 Lý do áp dụng

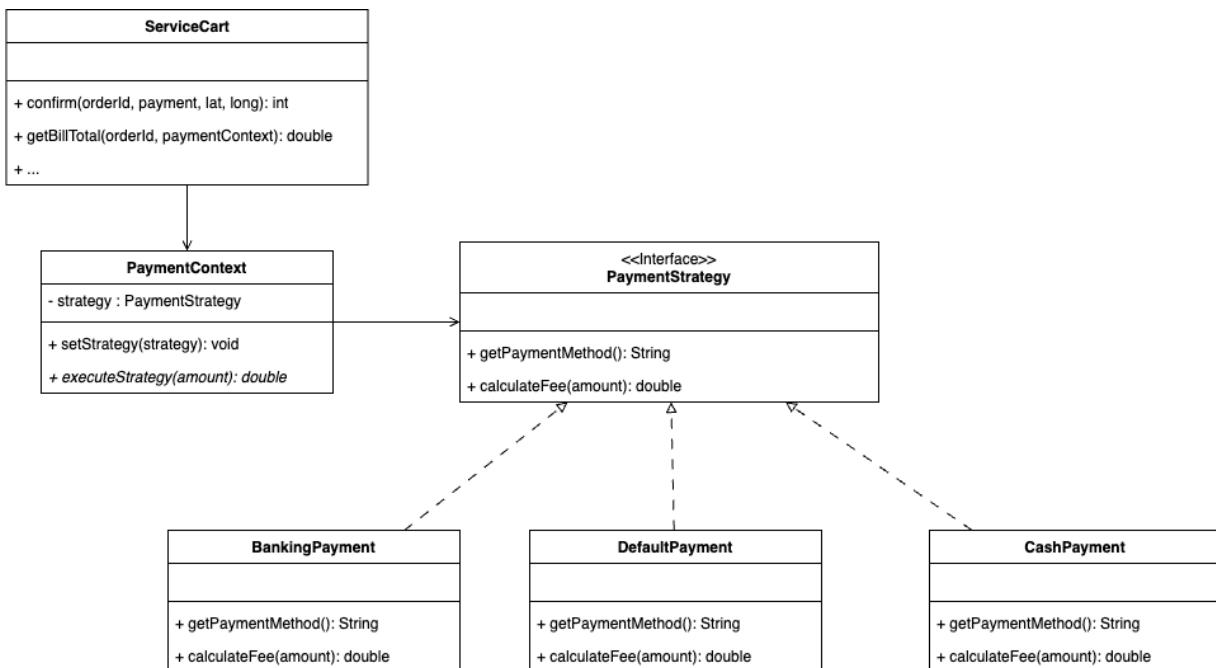
Lý do	Mô tả cụ thể theo code
Tách biệt thuật toán khỏi logic chính	Tính phí chuyển khoản, tiền mặt được đóng gói riêng trong các class BankingPayment, CashPayment...
Dễ mở rộng	Nếu bạn muốn thêm thanh toán QR code hoặc ví điện tử, chỉ cần tạo class mới mà không cần sửa code trong ServiceCart.

Tuân thủ nguyên tắc OCP (Open/Closed Principle)	Hệ thống mở rộng dễ dàng mà không ảnh hưởng logic hiện tại.
Linh hoạt và thay đổi tại runtime	Bạn có thể thay đổi PaymentStrategy ở bất cứ thời điểm nào, mà không ảnh hưởng đến PaymentContext.
Tái sử dụng chiến lược	calculateFee() có thể tái sử dụng ở nhiều nơi khác nhau (không chỉ trong confirm mà còn hiển thị trước khi thanh toán chẳng hạn).

Bảng 2. 6 Lý do áp dụng Strategy Pattern

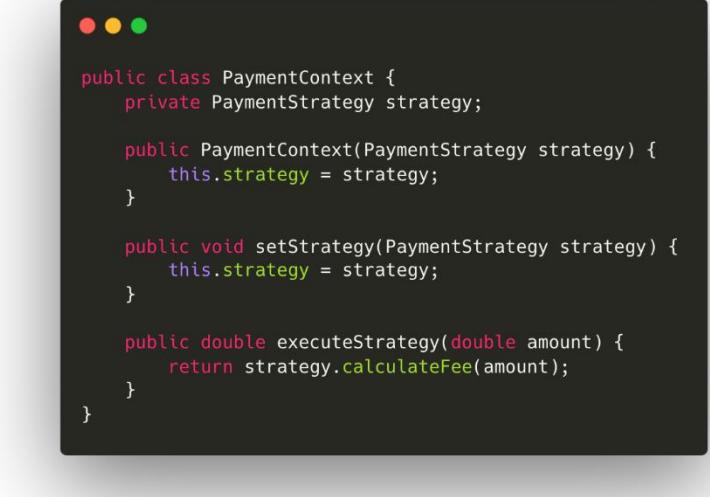
Tóm lại: Áp dụng Strategy Pattern để linh hoạt thay đổi cách tính phí thanh toán (tiền mặt, chuyển khoản...) mà không ảnh hưởng đến logic xử lý đơn hàng trong ServiceCart, giúp code dễ bảo trì, mở rộng và tuân thủ nguyên tắc SOLID

2.4.2 Sơ đồ lớp



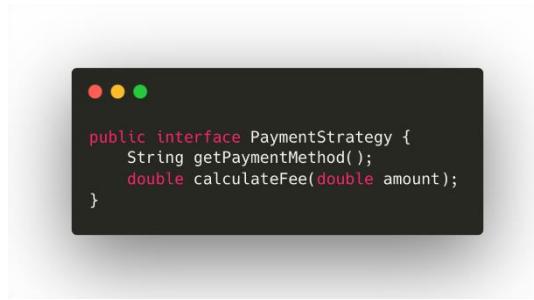
Hình 2. 18 Sơ đồ lớp Payment áp dụng Strategy Pattern

2.4.3 Code áp dụng



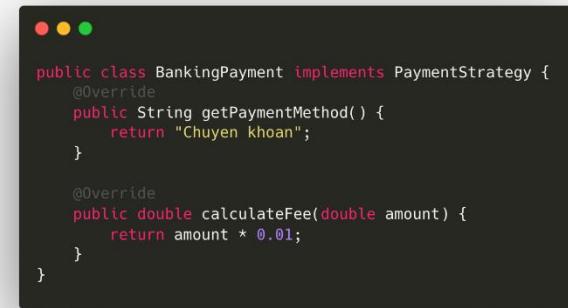
```
public class PaymentContext {  
    private PaymentStrategy strategy;  
  
    public PaymentContext(PaymentStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void setStrategy(PaymentStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public double executeStrategy(double amount) {  
        return strategy.calculateFee(amount);  
    }  
}
```

Hình 2. 19 Lớp PaymentContext



```
public interface PaymentStrategy {  
    String getPaymentMethod();  
    double calculateFee(double amount);  
}
```

Hình 2. 20 Lớp PaymentStrategy



```
public class BankingPayment implements PaymentStrategy {  
    @Override  
    public String getPaymentMethod() {  
        return "Chuyen khoan";  
    }  
  
    @Override  
    public double calculateFee(double amount) {  
        return amount * 0.01;  
    }  
}
```

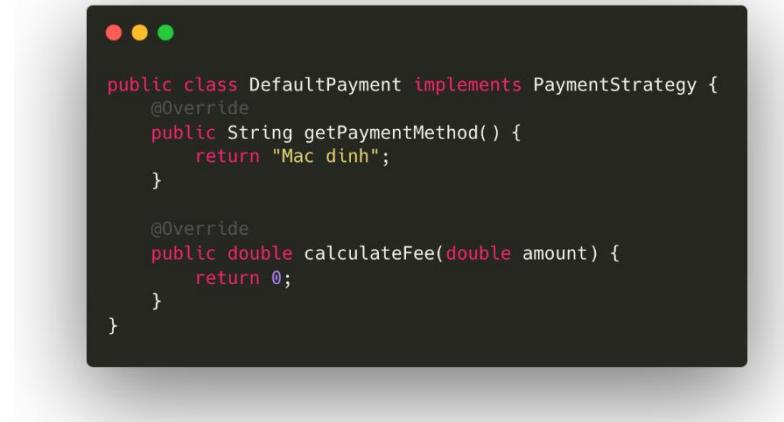
Hình 2. 21 Lớp BankingPayment



```
public class CashPayment implements PaymentStrategy {
    @Override
    public String getPaymentMethod() {
        return "Tien mat";
    }

    @Override
    public double calculateFee(double amount) {
        return 0;
    }
}
```

Hình 2. 22 Lớp CashPayment



```
public class DefaultPayment implements PaymentStrategy {
    @Override
    public String getPaymentMethod() {
        return "Mac dinh";
    }

    @Override
    public double calculateFee(double amount) {
        return 0;
    }
}
```

Hình 2. 23 Lớp DefaultPayment

```
@Service
public class ServiceCart {
    ...
    public int confirm(String order_id, int payment, double latitude, double longitude) {
        try {
            connection = dbConnector.connect();
            String query = "{CALL Confirm(?, ?, ?, ?)}";
            callableStatement = connection.prepareCall(query);
            callableStatement.setString(1, order_id);

            //Khởi tạo mặc định
            PaymentContext context = new PaymentContext(new DefaultPayment());

            // Khởi tạo đối tượng tương ứng ( new BankingPayment() ; new CashPayment() )
            PaymentStrategy paymentStrategy = getPaymentStrategy(payment);

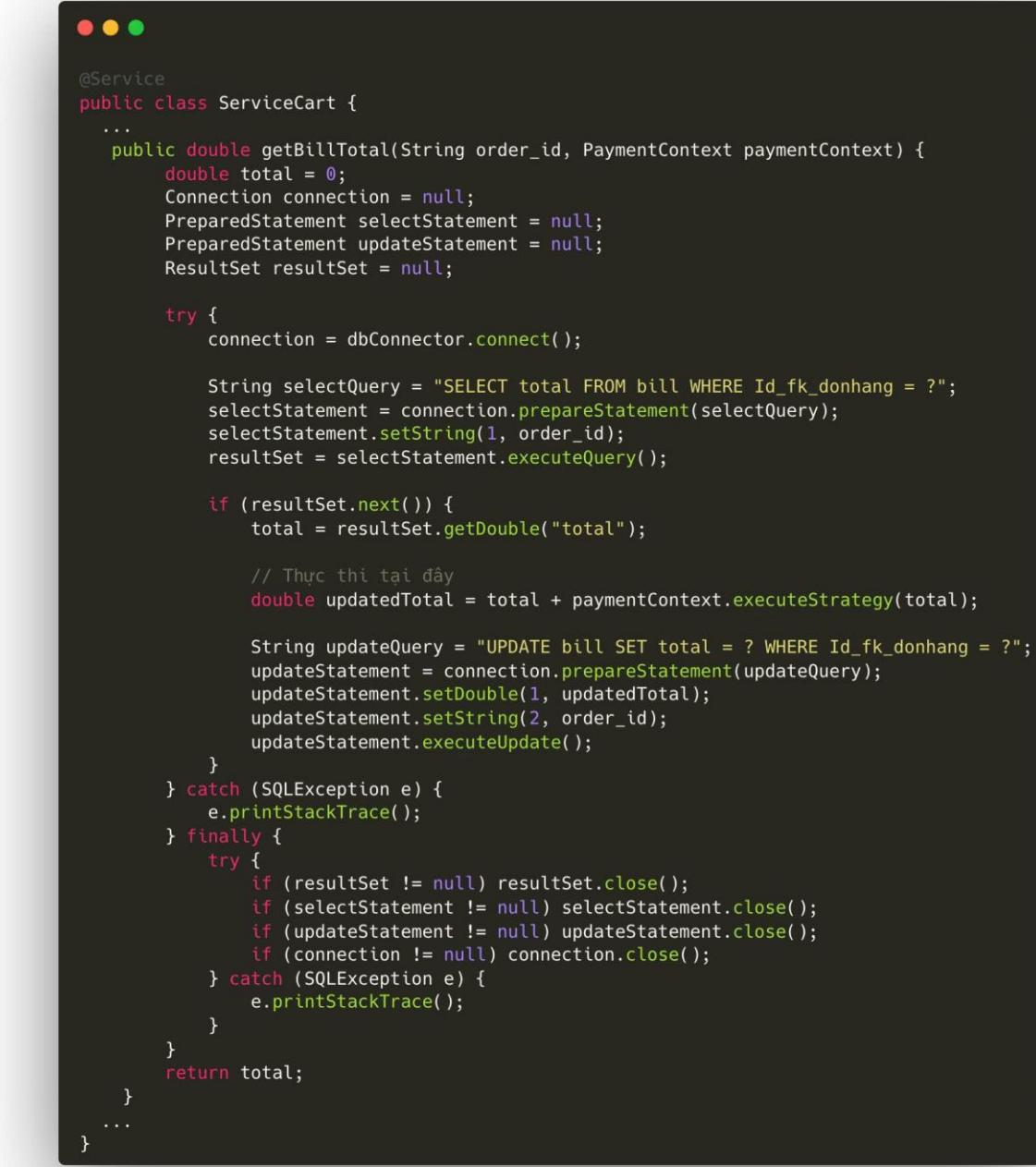
            // Thay đổi đối tượng cần thực thi
            context.setStrategy(paymentStrategy);

            if (paymentStrategy == null) {
                throw new IllegalArgumentException("Phương thức thanh toán không hợp lệ");
            }

            String method = paymentStrategy.getPaymentMethod();
            callableStatement.setString(2, method);
            callableStatement.setDouble(3, latitude);
            callableStatement.setDouble(4, longitude);
            callableStatement.execute();

            getBillTotal(order_id, context);
            return 1;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }

    private PaymentStrategy getPaymentStrategy(int payment) {
        switch (payment) {
            case 1:
                return new BankingPayment();
            case 2:
                return new CashPayment();
            default:
                return null;
        }
    }
    ...
}
```



The screenshot shows a Java code editor with a dark theme. At the top left are three colored window control buttons (red, yellow, green). The code itself is a Java class named `ServiceCart`. It includes annotations `@Service` and `...`, and contains several methods and variable declarations. The code is primarily focused on performing database operations using JDBC, specifically selecting a total value from a `bill` table where `Id_fk_donhang` matches a given `order_id`, and then updating the `total` value in the same table. Error handling is provided via `SQLException` catches and stack trace prints. Finally, resources are cleaned up by closing statements and connections.

```

@Service
public class ServiceCart {
    ...
    public double getBillTotal(String order_id, PaymentContext paymentContext) {
        double total = 0;
        Connection connection = null;
        PreparedStatement selectStatement = null;
        PreparedStatement updateStatement = null;
        ResultSet resultSet = null;

        try {
            connection = dbConnector.connect();

            String selectQuery = "SELECT total FROM bill WHERE Id_fk_donhang = ?";
            selectStatement = connection.prepareStatement(selectQuery);
            selectStatement.setString(1, order_id);
            resultSet = selectStatement.executeQuery();

            if (resultSet.next()) {
                total = resultSet.getDouble("total");

                // Thực thi tại đây
                double updatedTotal = total + paymentContext.executeStrategy(total);

                String updateQuery = "UPDATE bill SET total = ? WHERE Id_fk_donhang = ?";
                updateStatement = connection.prepareStatement(updateQuery);
                updateStatement.setDouble(1, updatedTotal);
                updateStatement.setString(2, order_id);
                updateStatement.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (resultSet != null) resultSet.close();
                if (selectStatement != null) selectStatement.close();
                if (updateStatement != null) updateStatement.close();
                if (connection != null) connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return total;
    }
    ...
}

```

Hình 2. 24 Lớp ServiceCart

2.5 Command Pattern

2.5.1 Lý do áp dụng

2.5.1.1 Khái niệm

Command Pattern là một mẫu thiết kế thuộc nhóm Creational Design Patterns dùng để đóng gói một yêu cầu dưới dạng đối tượng. Mẫu thiết kế này cho phép ta tách biệt việc thực hiện một hành động khỏi việc gọi hành động đó. Điều này giúp hệ thống linh hoạt hơn, dễ dàng mở rộng và thay đổi, đồng thời hỗ trợ các tính năng như hoàn tác (undo), lặp lại (redo) hoặc xử lý lệnh bất đồng bộ.

2.5.1.2 Lý do áp dụng

Yếu tố	Lý do áp dụng Command Pattern
Tách biệt hành động và yêu cầu	Command Pattern cho phép tách biệt yêu cầu hành động (add, minus, delete...) khỏi đối tượng thực hiện hành động. Điều này giúp mã dễ bảo trì và mở rộng, đặc biệt khi có thêm yêu cầu mới trong tương lai.
Dễ dàng mở rộng hệ thống	Mỗi hành động như thêm món vào giỏ, xóa món khỏi giỏ, hay thay đổi số lượng đều được đóng gói dưới dạng các command khác nhau (AddToCartCommand, MinusToCartCommand, v.v.). Khi có yêu cầu mới, ta chỉ cần tạo thêm command mà không cần thay đổi logic hiện tại.
Lưu trữ và thực thi lại hành động	Các lệnh đã thực thi có thể được lưu lại trong CartInvoker, giúp ta quản lý lịch sử lệnh hoặc hoàn tác (undo) khi cần thiết (mặc dù trong mã của ta không thấy yêu cầu hoàn tác nhưng nó vẫn dễ dàng mở rộng trong tương lai).
Hỗ trợ cho các yêu cầu bất đồng bộ	Việc sử dụng executeCommand trong CartInvoker giúp tách biệt logic thực thi và gọi lệnh, tạo điều kiện cho việc

	xử lý bất đồng bộ, đặc biệt là khi hệ thống có thể cần xử lý các yêu cầu từ nhiều người dùng cùng lúc.
Sử dụng như một hệ thống lệnh linh hoạt	Việc sử dụng ICommand giúp tạo ra một hệ thống linh hoạt, nơi mỗi hành động cụ thể (thêm món, xóa món, v.v.) có thể được đóng gói dưới dạng một lệnh độc lập, dễ dàng thay đổi hoặc mở rộng mà không ảnh hưởng đến các phần khác của hệ thống.
Kiểm tra và bảo trì dễ dàng	Mỗi lệnh (Command) có thể được kiểm tra độc lập, vì vậy nếu có sự cố xảy ra, ta dễ dàng phát hiện lỗi và sửa chữa mà không làm gián đoạn phần khác trong ứng dụng. Ví dụ, có thể dễ dàng kiểm tra AddToCartCommand mà không cần phải quan tâm đến toàn bộ ứng dụng.
Sử dụng tốt cho các hành động nhiều bước	Nếu muốn một lệnh thực hiện các bước nhiều thao tác hoặc các thao tác phức tạp, Command Pattern cho phép dễ dàng quản lý các bước và tổ chức lại hành động mà không làm tăng độ phức tạp của lớp gọi lệnh (controller).

Bảng 2. 7 Lý do áp dụng Command Pattern

2.5.1.3 Phân tích vai trò các lớp

Bảng 2. 8 Các thành phần của Command Pattern

Các thành phần trong lớp tổng quát	Cụ thể
Invoker	CartInvoker
Receiver	ServiceCart
Command	ICommand
ConcreteCommand	AddToCartCommand, AddMoreToCartCommand,

	DeleteFromCartCommand, MinusToCartCommand
Client	ControllerCart

ICommand (Interface): Định nghĩa phương thức execute() để các lớp lệnh cụ thể thực thi hành động. Đây là interface chung mà tất cả các lệnh đều phải tuân theo, giúp đảm bảo tính đồng nhất.

Các lớp lệnh cụ thể (Ví dụ: AddToCartCommand, MinusToCartCommand, DeleteFromCartCommand): Mỗi lớp này đóng gói một hành động cụ thể như thêm món vào giỏ, xóa món khỏi giỏ, thay đổi số lượng món. Các lớp này thực hiện lệnh thông qua phương thức execute() của interface ICommand.

CartInvoker: Lớp này quản lý việc thực thi các lệnh. Nó nhận các lệnh từ bên ngoài và gọi phương thức execute() của từng lệnh. Đây là nơi thực hiện các hành động và có thể lưu trữ các lệnh để thực thi lại hoặc hoàn tác (undo) nếu cần.

Cart: Lớp này đại diện cho đối tượng giỏ hàng và chứa các phương thức thao tác trực tiếp với dữ liệu giỏ hàng, như thêm/xóa món. Nó được sử dụng trong các lệnh cụ thể để thực hiện thay đổi đối tượng giỏ hàng.

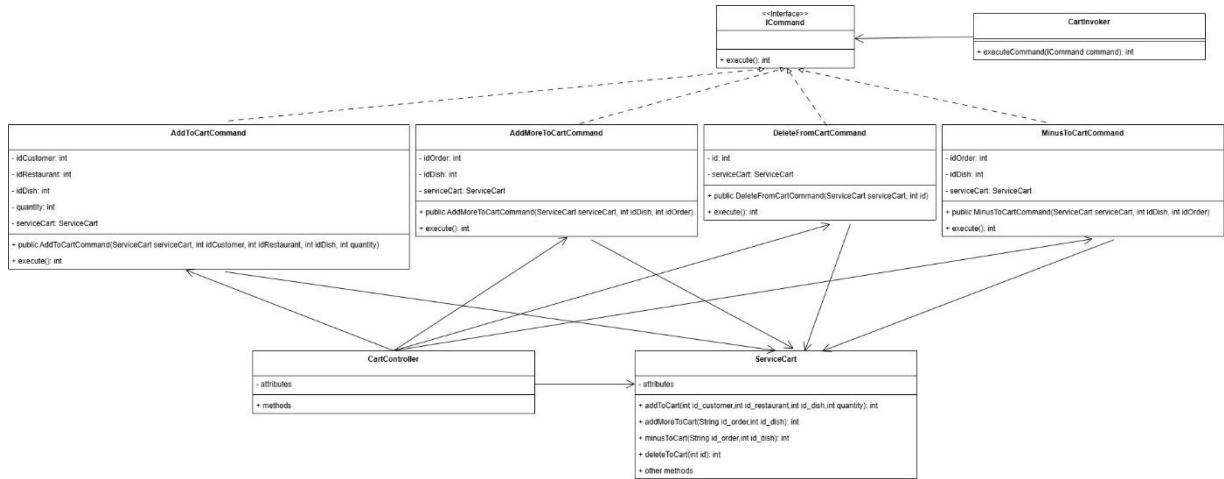
CartController: Đây là lớp trung gian giữa giao diện người dùng và các lệnh. Nó nhận các yêu cầu từ người dùng và sử dụng CartInvoker để gọi các lệnh thích hợp, giúp tách biệt logic xử lý giao diện với các thao tác kinh doanh (thêm/xóa món).

Tóm lại:

- ICommand: Định nghĩa các hành động.
- Lớp lệnh cụ thể: Thực hiện các hành động cụ thể.
- CartInvoker: Quản lý và thực thi các lệnh.
- Cart: Thực hiện thao tác trên giỏ hàng.
- CartController: Xử lý yêu cầu từ người dùng và gọi lệnh phù hợp.

Mỗi lớp trong mô hình Command Pattern giúp tách biệt rõ ràng các nhiệm vụ và hành động trong ứng dụng, giúp dễ bảo trì, mở rộng và kiểm thử.

2.5.2 Sơ đồ lớp



Hình 2. 25 Sơ đồ lớp áp dụng Command Pattern vào các Command của Cart

2.5.3 Code áp dụng

```

package
com.example.Api.pattern.command;
public interface ICommand {
    int execute();
}
  
```

Hình 2. 26 Lớp ICommand

```
package com.example.Api.pattern.command;

import java.util.ArrayList;
import java.util.List;

public class CartInvoker {
    private List< ICommand> commandHistory = new ArrayList<>();
    public int executeCommand(ICommand command) {
        commandHistory.add(command);
        return command.execute();
    }
}
```

Hình 2. 27 Lớp CartInvoker

```
package com.example.Api.pattern.command;

import com.example.Api.service.ServiceCart;

public class AddMoreToCartCommand implements ICommand {
    private String idOrder;
    private int idDish;
    private ServiceCart serviceCart;

    public AddMoreToCartCommand(ServiceCart serviceCart, String idOrder, int idDish)
    {
        this.serviceCart = serviceCart;
        this.idOrder = idOrder;
        this.idDish = idDish;
    }

    @Override
    public int execute() {
        return serviceCart.addMoreToCart(idOrder, idDish);
    }
}
```

Hình 2. 28 Lớp AddMoreToCartCommand

```
● ● ●

package com.example.Api.pattern.command;
import com.example.Api.service.ServiceCart;

public class AddToCartCommand implements ICommand {
    private int idCustomer;
    private int idRestaurant;
    private int idDish;
    private int quantity;
    private ServiceCart serviceCart;

    public AddToCartCommand(ServiceCart serviceCart, int idCustomer, int idRestaurant, int idDish, int
    quantity) {
        this.serviceCart = serviceCart;
        this.idCustomer = idCustomer;
        this.idRestaurant = idRestaurant;
        this.idDish = idDish;
        this.quantity = quantity;
    }

    @Override
    public int execute() {
        return serviceCart.addToCart(idCustomer, idRestaurant, idDish, quantity);
    }
}
```

Hình 2. 29 Lớp AddToCartCommand

```
● ● ●

package com.example.Api.pattern.command;

import com.example.Api.service.ServiceCart;

public class DeleteFromCartCommand implements ICommand {
    private int id;
    private ServiceCart serviceCart;

    public DeleteFromCartCommand(ServiceCart serviceCart, int id) {
        this.serviceCart = serviceCart;
        this.id = id;
    }

    @Override
    public int execute() {
        return serviceCart.deleteToCart(id);
    }
}
```

Hình 2. 30 Lớp DeleteFromCartCommand

```
● ● ●  
package com.example.Api.pattern.command;  
  
import com.example.Api.service.ServiceCart;  
  
public class MinusToCartCommand implements ICommand {  
    private String idOrder;  
    private int idDish;  
    private ServiceCart serviceCart;  
  
    public MinusToCartCommand(ServiceCart serviceCart, String idOrder, int idDish) {  
        this.serviceCart = serviceCart;  
        this.idOrder = idOrder;  
        this.idDish = idDish;  
    }  
  
    @Override  
    public int execute() {  
        return serviceCart.MinusToCart(idOrder, idDish);  
    }  
}
```

Hình 2. 31 Lớp MinusToCartCommand



```

package com.example.Api.controller;

import com.example.Api.model.OrderDetail;
import com.example.Api.model.Restaurant;
import com.example.Api.pattern.command.*;
import com.example.Api.service.ServiceCart;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/cart")
public class ControllerCart {
    private final CartInvoker invoker = new CartInvoker();
    private ServiceCart serviceCart;

    public ControllerCart(ServiceCart serviceCart) {
        this.serviceCart = serviceCart;
    }

    @PostMapping("/add")
    public ResponseEntity<String> addDishToCart(@RequestParam("id_customer") int id_customer,
                                                @RequestParam("id_restaurant") int id_restaurant,
                                                @RequestParam("id_dish") int id_dish,
                                                @RequestParam("quantity") int quantity) {
        ICommand addToCartCommand = new AddToCartCommand(serviceCart, id_customer, id_restaurant,
                                                       id_dish, quantity);
        int result = invoker.executeCommand(addToCartCommand);
        if (result == 1) {
            return ResponseEntity.ok("Add to cart success.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Failed.");
        }
    }

    @PostMapping("/addMore")
    public ResponseEntity<String> addMore(@RequestParam("id_order") String id_order,
                                           @RequestParam("id_dish") int id_dish) {
        ICommand command = new AddMoreToCartCommand(serviceCart, id_order, id_dish);
        int result = invoker.executeCommand(command);
        if (result == 1) {
            return ResponseEntity.ok("Add more to cart success.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Failed.");
        }
    }

    @PostMapping("/minus")
    public ResponseEntity<String> minus(@RequestParam("id_order") String id_order,
                                         @RequestParam("id_dish") int id_dish) {
        ICommand command = new MinusToCartCommand(serviceCart, id_order, id_dish);
        int result = invoker.executeCommand(command);
        if (result == 1) {
            return ResponseEntity.ok("Minus to cart success.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Failed.");
        }
    }

    @PostMapping("/delete")
    public ResponseEntity<String> deleteToCart(@RequestParam("id") int id) {
        ICommand command = new DeleteFromCartCommand(serviceCart, id);
        int result = invoker.executeCommand(command);
        if (result == 1) {
            return ResponseEntity.ok("Delete to cart success.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Failed.");
        }
    }

    // Other methods
}

```

Hình 2. 32 Lớp ControllerCart

2.6 Observer Pattern

2.6.1 Lý do áp dụng

2.6.1.1 Khái niệm Observer Pattern

- **Observer Pattern** là một mẫu thiết kế thuộc nhóm Hành vi (**Behavioral Pattern**).
- Nó định nghĩa mối quan hệ một-nhiều giữa các đối tượng: Khi một đối tượng (subject) thay đổi trạng thái, tất cả các observer của nó sẽ được thông báo và tự động cập nhật.

2.6.1.2 Các thành phần trong code

Vai trò trong Observer Pattern	Lớp tương ứng trong code
Subject (Chủ đề)	OrderManager
Observer (Người quan sát)	CustomerOrderFood (implements OrderObserver)
Giao diện Subject	ISubject
Giao diện Observer	OrderObserver
Sự kiện để thông báo	Gọi observer.onOrdersUpdated(...) trong onResponse()

Bảng 2. 9 Các thành phần trong Observer Pattern

2.6.1.3 Lý do áp dụng Observer Pattern trong ứng dụng

Khi danh sách đơn hàng được lấy thành công, các thành phần khác (UI, màn hình, v.v.) **phải được cập nhật lập tức**, mà không cần subject biết cụ thể chúng là gì.

2.6.1.4 Lợi ích cụ thể

Lợi ích	Giải thích theo code
Tách biệt logic xử lý và hiển thị	OrderManager chỉ lo gọi API và thông báo; CustomerOrderFood lo hiển thị Toast.
Dễ mở rộng	Muốn thêm màn hình khác nhận update? → chỉ cần implement OrderObserver.
Tăng tính linh hoạt	OrderManager không cần biết ai đang lắng nghe, chỉ cần gọi observer.onOrdersUpdated(...).

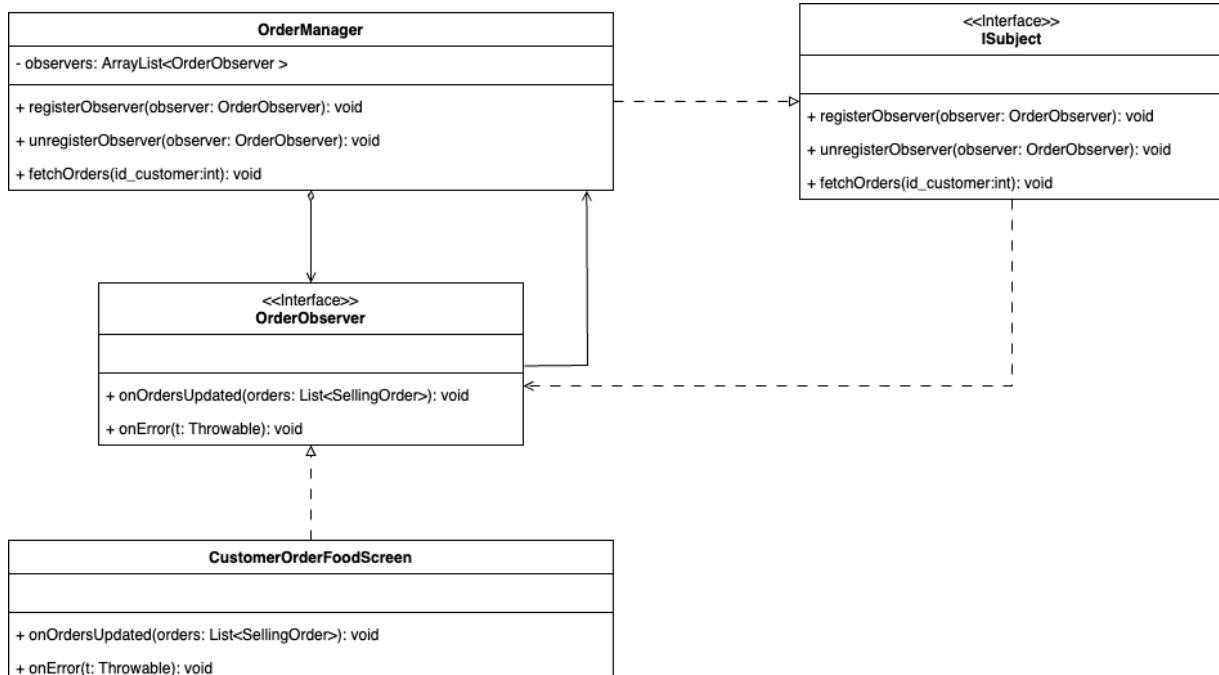
Phản ứng tự động với dữ liệu mới	Khi đơn hàng được cập nhật, UI được tự động phản hồi thay vì phải kiểm tra thủ công.
Giảm sự phụ thuộc (Low coupling)	OrderManager và CustomerOrderFood chỉ liên kết qua interface OrderObserver. Không có liên kết trực tiếp.

Bảng 2. 10 Lý do áp dụng Observer Pattern

2.6.1.5 Tình huống áp dụng phù hợp

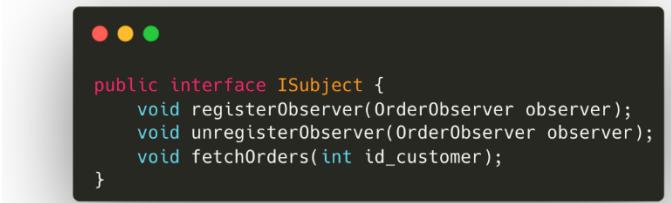
- OrderManager đóng vai trò trung tâm xử lý đơn hàng.
- Khi đơn hàng được xác nhận thành công → cập nhật giao diện (hiển thị số lượng đơn hàng mới...) → nên dùng Observer để xử lý tự động và linh hoạt.
- Tóm lại: Áp dụng Observer Pattern để cho phép OrderManager thông báo đến nhiều đối tượng giao diện (như CustomerOrderFood) mỗi khi có dữ liệu đơn hàng mới, giúp cập nhật UI tự động, linh hoạt và giảm phụ thuộc giữa các thành phần.

2.6.2 Sơ đồ lớp



Hình 2. 33 Sơ đồ lớp áp dụng Observer Pattern vào Order

2.6.3 Code áp dụng

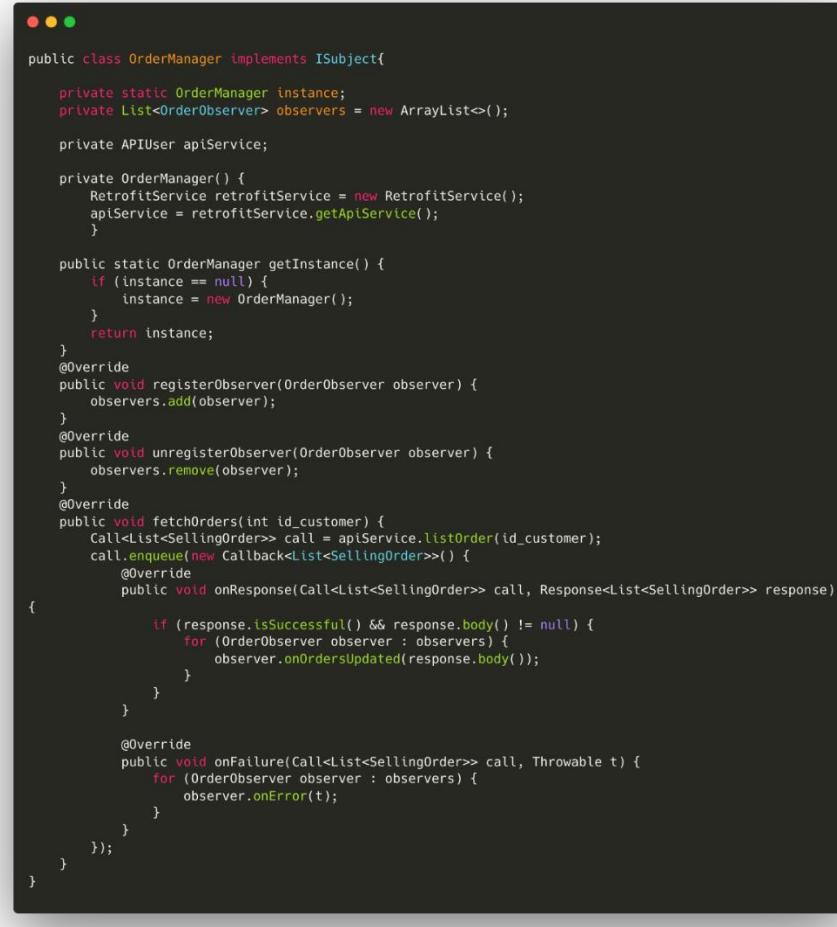


```

public interface ISubject {
    void registerObserver(OrderObserver observer);
    void unregisterObserver(OrderObserver observer);
    void fetchOrders(int id_customer);
}

```

Hình 2. 34 Lớp ISubject



```

public class OrderManager implements ISubject{
    private static OrderManager instance;
    private List<OrderObserver> observers = new ArrayList<>();

    private APIUser apiService;

    private OrderManager() {
        RetrofitService retrofitService = new RetrofitService();
        apiService = retrofitService.getAPIService();
    }

    public static OrderManager getInstance() {
        if (instance == null) {
            instance = new OrderManager();
        }
        return instance;
    }

    @Override
    public void registerObserver(OrderObserver observer) {
        observers.add(observer);
    }

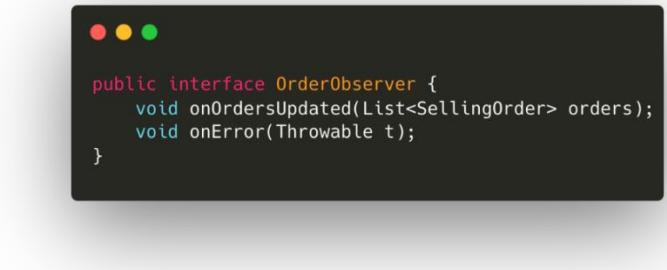
    @Override
    public void unregisterObserver(OrderObserver observer) {
        observers.remove(observer);
    }

    @Override
    public void fetchOrders(int id_customer) {
        Call<List<SellingOrder>> call = apiService.listOrder(id_customer);
        call.enqueue(new Callback<List<SellingOrder>>() {
            @Override
            public void onResponse(Call<List<SellingOrder>> call, Response<List<SellingOrder>> response) {
                if (response.isSuccessful() && response.body() != null) {
                    for (OrderObserver observer : observers) {
                        observer.onOrdersUpdated(response.body());
                    }
                }
            }

            @Override
            public void onFailure(Call<List<SellingOrder>> call, Throwable t) {
                for (OrderObserver observer : observers) {
                    observer.onError(t);
                }
            }
        });
    }
}

```

Hình 2. 35 Lớp Order Manager



Hình 2. 36 Lớp OrderObserver

```

public class CustomerOrderFood extends AppCompatActivity implements OrderObserver {
    ...
    @Override protected void onCreate(Bundle savedInstanceState) {
        ...
        button6.setOnClickListener(new View.OnClickListener() {
            @Override public void onClick(View view) {
                if (!position_now.isChecked()) {
                    Toast
                        .makeText(CustomerOrderFood.this,
                                "Vui lòng chọn địa điểm nhận hàng.", Toast.LENGTH_SHORT)
                        .show();
                    return;
                }

                if (checkPayment == -1) {
                    Toast
                        .makeText(CustomerOrderFood.this,
                                "Vui lòng chọn phương thức thanh toán.", Toast.LENGTH_SHORT)
                        .show();
                    return;
                }
                if (check == 0) {
                    Toast
                        .makeText(CustomerOrderFood.this,
                                "Vui lòng cập nhật địa chỉ và số điện thoại.", Toast.LENGTH_SHORT)
                        .show();
                    return;
                }

                Call<String> call = retrofitService.getApiService().confirm(
                    orderId, checkPayment, lati, longi);
                call.enqueue(new Callback<String>() {
                    @Override public void onResponse(Call<String> call,
                        Response<String> response) {
                        if (response.isSuccessful()) {
                            Toast
                                .makeText(CustomerOrderFood.this, "Đặt đơn thành công",
                                        Toast.LENGTH_LONG)
                                .show();
                            OrderManager.getInstance().registerObserver(
                                CustomerOrderFood.this); // Sử dụng tại đây
                            OrderManager.getInstance().fetchOrders(id);
                            finish();

                        } else {
                            Toast
                                .makeText(CustomerOrderFood.this,
                                        "Error: " + response.message(), Toast.LENGTH_SHORT)
                                .show();
                        }
                    }

                    @Override public void onFailure(Call<String> call, Throwable t) {
                    }
                });
            }
        });
    }

    @Override public void onOrdersUpdated(List<SellingOrder> orders) {
        Toast
            .makeText(this, "Bạn đang có " + orders.size() + " đơn hàng!",
                    Toast.LENGTH_SHORT)
            .show();
    }

    @Override public void onError(Throwable t) {
        Toast
            .makeText(this, "Lỗi khi tải đơn hàng: " + t.getMessage(),
                    Toast.LENGTH_LONG)
            .show();
    }
}

```

Hình 2. 37 Lớp CustomerOrderFood

2.7 Adapter Pattern

2.7.1 Lý do áp dụng

2.7.1.1 Khái niệm

Adapter Pattern là một thiết kế mẫu cấu trúc (structural design pattern) dùng để chuyển đổi giao diện của một lớp thành một giao diện mà client mong muốn. Nó cho phép các lớp có giao diện không tương thích với nhau có thể làm việc cùng nhau mà không cần thay đổi mã nguồn của các lớp đó.

2.7.1.2 Lý do áp dụng

Mục tiêu	Lý do áp dụng Adapter Pattern
Khả năng mở rộng	Adapter Pattern giúp hệ thống dễ dàng mở rộng trong tương lai mà không làm ảnh hưởng đến các phần khác của ứng dụng. Ví dụ, nếu ta muốn chuyển sang một dịch vụ gửi email khác hoặc thêm nhiều dịch vụ gửi email (như dịch vụ qua API thay vì SMTP), ta chỉ cần tạo một adapter mới mà không phải sửa đổi các phần còn lại của hệ thống. Điều này tăng tính linh hoạt và giảm sự phụ thuộc vào chi tiết cụ thể của từng dịch vụ gửi email. Ta có thể thêm các adapter mới mà không cần thay đổi mã nguồn của các lớp sử dụng giao diện IEmail.
Tách biệt giao diện và logic cụ thể	Adapter Pattern giúp tách biệt giao diện gửi email (IEmail) với hệ thống gửi email thực tế (ISmtp), giảm sự phụ thuộc vào cách thức gửi email cụ thể. Người dùng giao diện IEmail không cần phải biết các chi tiết của việc gửi email thông qua SmtpAdaptee.

Đảm bảo tính tương thích giữa các hệ thống	EmailAdapter chuyển giao diện từ IEmail sang ISmtpt, giúp các hệ thống khác trong ứng dụng có thể gửi email mà không cần biết chi tiết của hệ thống gửi email, do đó dễ dàng sử dụng và thay thế các phương thức gửi email mà không gây gián đoạn.
Giảm thiểu sự thay đổi mã nguồn hiện tại	Nếu muốn thay đổi hệ thống gửi email (chuyển từ SMTP sang API khác), chỉ cần thay đổi trong SmtpAdaptee mà không làm ảnh hưởng đến các lớp sử dụng giao diện IEmail. Điều này giúp duy trì tính ổn định của ứng dụng khi có sự thay đổi hệ thống gửi email.
Dễ dàng tích hợp với các hệ thống khác	Với Adapter Pattern, việc thay đổi hoặc tích hợp các hệ thống gửi email mới trở nên dễ dàng mà không làm ảnh hưởng đến các thành phần khác của ứng dụng, giúp dễ dàng hỗ trợ nhiều dịch vụ email mà không cần phải sửa mã nguồn của các lớp sử dụng giao diện chung IEmail.

Bảng 2. 11 Lý do áp dụng Adapter Pattern

2.7.1.3 Phân tích vai trò các lớp

Bảng 2. 12 Thành phần của Adapter Pattern

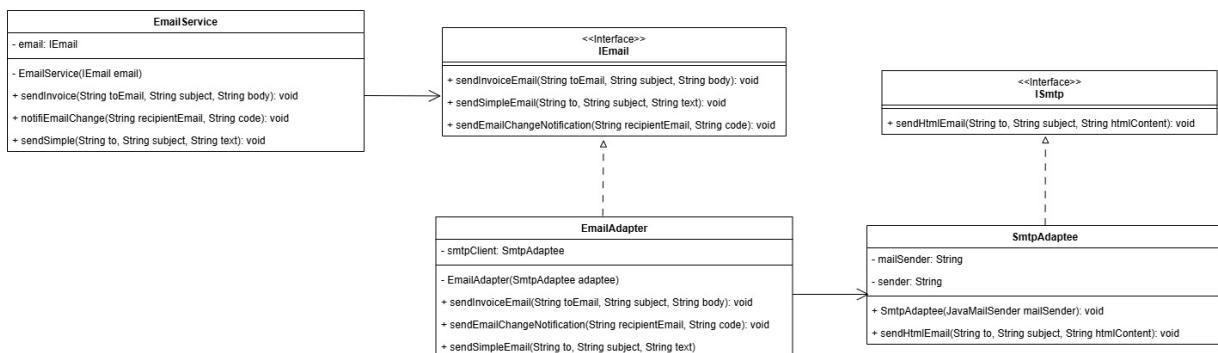
Thành phần trong lớp tổng quát	Cụ thể
Client	EmailService
Target	IEmail
Adapter	EmailAdapter
Adaptee	SmtpAdaptee

Tóm tắt vai trò:

- EmailAdapter làm cầu nối giữa giao diện chung và hệ thống gửi email thực tế.

- SmtpAdaptee thực hiện công việc gửi email.
- IEmail là giao diện chung mà các lớp khác trong ứng dụng sử dụng.
- ISmtip là giao diện thực tế của hệ thống gửi email qua SMTP.
- ControllerSendEmail nhận yêu cầu từ người dùng và gửi email qua EmailService.
- EmailService xử lý nghiệp vụ và gọi EmailAdapter để gửi email.

2.7.2 Sơ đồ lớp



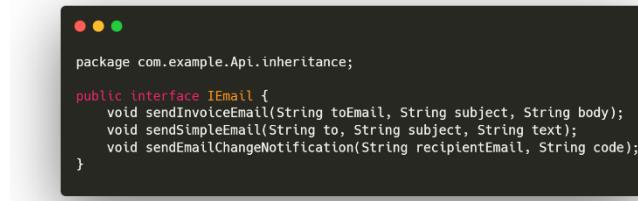
Hình 2. 38 Sơ đồ lớp áp dụng Adapter Pattern vào Email Service

2.7.3 Code áp dụng

```

  package com.example.Api.inheritance;
  import jakarta.mail.MessagingException;
  public interface ISmtip {
      public void sendHtmlEmail(String to, String subject, String htmlContent) throws MessagingException;
  }
  
```

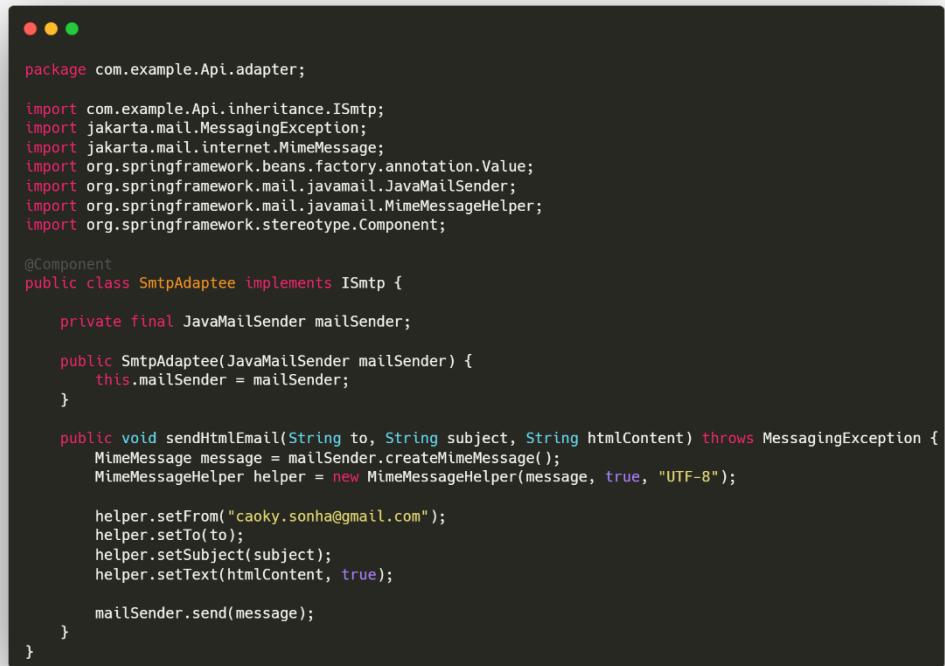
Hình 2. 39 Lớp ISmtip



```
package com.example.Api.inheritance;

public interface IEmail {
    void sendInvoiceEmail(String toEmail, String subject, String body);
    void sendSimpleEmail(String to, String subject, String text);
    void sendEmailChangeNotification(String recipientEmail, String code);
}
```

Hình 2. 40 Lớp IEmail



```
package com.example.Api.adapter;

import com.example.Api.inheritance.ISsmtp;
import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Component;

@Component
public class SmtpAdaptee implements ISsmtp {

    private final JavaMailSender mailSender;

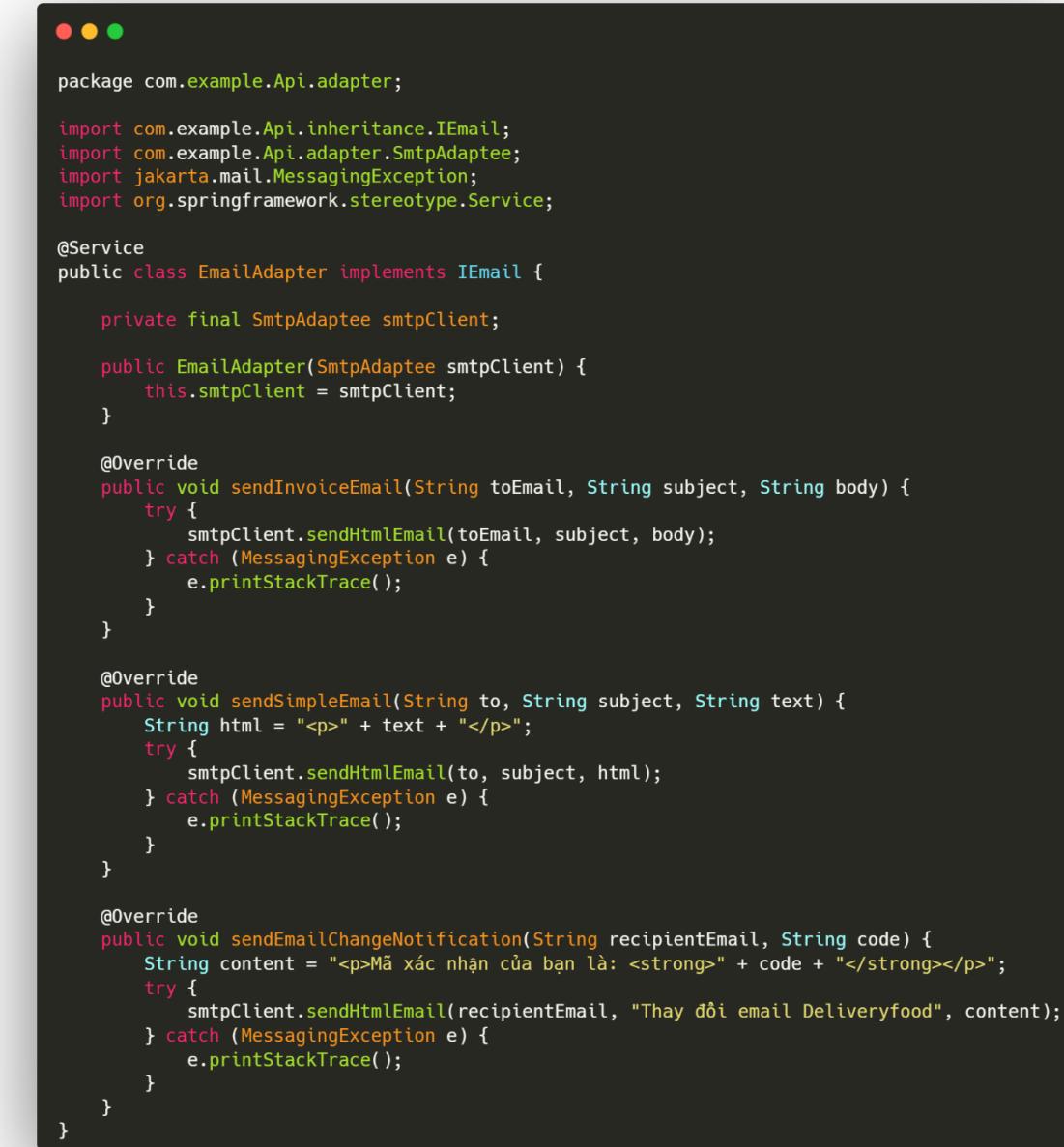
    public SmtpAdaptee(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void sendHtmlEmail(String to, String subject, String htmlContent) throws MessagingException {
        MimeMessage message = mailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message, true, "UTF-8");

        helper.setFrom("caoky.sonha@gmail.com");
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(htmlContent, true);

        mailSender.send(message);
    }
}
```

Hình 2. 41 Lớp SmtpAdaptee



```
package com.example.Api.adapter;

import com.example.Api.inheritance.IEmail;
import com.example.Api.adapter.SmtpAdaptee;
import jakarta.mail.MessagingException;
import org.springframework.stereotype.Service;

@Service
public class EmailAdapter implements IEmail {

    private final SmtpAdaptee smtpClient;

    public EmailAdapter(SmtpAdaptee smtpClient) {
        this.smtpClient = smtpClient;
    }

    @Override
    public void sendInvoiceEmail(String toEmail, String subject, String body) {
        try {
            smtpClient.sendHtmlEmail(toEmail, subject, body);
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void sendSimpleEmail(String to, String subject, String text) {
        String html = "<p>" + text + "</p>";
        try {
            smtpClient.sendHtmlEmail(to, subject, html);
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void sendEmailChangeNotification(String recipientEmail, String code) {
        String content = "<p>Mã xác nhận của bạn là: <strong>" + code + "</strong></p>";
        try {
            smtpClient.sendHtmlEmail(recipientEmail, "Thay đổi email Deliveryfood", content);
        } catch (MessagingException e) {
            e.printStackTrace();
        }
    }
}
```

Hình 2. 42 Lớp Email Adapter



```

package com.example.Api.service;

import com.example.Api.inheritance.IEmail;
import jakarta.mail.MessagingException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmailService {

    private final IEmail emailAdapter;

    @Autowired
    public EmailService(IEmail emailAdapter) {
        this.emailAdapter = emailAdapter;
    }

    public void sendInvoice(String toEmail, String subject, String body) {
        emailAdapter.sendInvoiceEmail(toEmail, subject, body);
    }

    public void sendSimple(String to, String subject, String text) {
        emailAdapter.sendSimpleEmail(to, subject, text);
    }

    public void notifyEmailChange(String recipientEmail, String code) throws MessagingException {
        emailAdapter.sendEmailChangeNotification(recipientEmail, code);
    }
}

```

Hình 2. 43 Lớp EmailService

2.8 Facade Pattern

2.8.1 Lý do áp dụng

2.8.1.1 Khái niệm Facade Pattern

Facade Pattern là một **structural design pattern** (mẫu thiết kế cấu trúc) dùng để cung cấp **một interface đơn giản duy nhất** để truy cập vào **một hệ thống phức tạp gồm nhiều lớp con** (subsystems). Mục tiêu là giảm sự phụ thuộc và phức tạp cho phía client (ví dụ: controller, UI)

2.8.1.2 Trong code

Vai trò	Lớp cụ thể
---------	------------

Facade chính	VoucherFacade
Subsystems (các lớp xử lý thực tế)	ServiceVoucherRestaurant, ServiceVoucherSystem
Client sử dụng Facade	ControllerVoucherRestaurant, ControllerVoucherSystem

Bảng 2. 13 Các thành phần của Facade Pattern

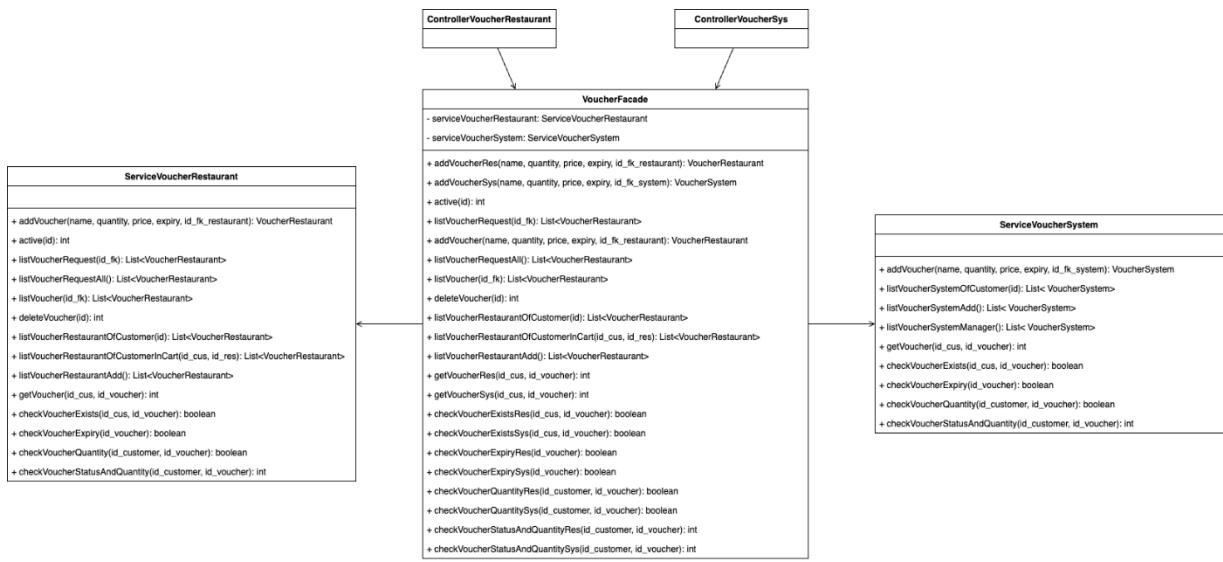
2.8.1.3 Lý do áp dụng Facade Pattern

Lý do	Mô tả theo code
Ẩn sự phức tạp	VoucherFacade ẩn đi việc xử lý kết nối DB, gọi từng ServiceVoucherRestaurant hoặc ServiceVoucherSystem.
Đơn giản hóa controller	Controller chỉ cần inject VoucherFacade mà không phải tạo nhiều service.
Dễ bảo trì	Nếu logic thay đổi trong service, controller không bị ảnh hưởng.
Tăng tính tái sử dụng	VoucherFacade có thể dùng lại ở nhiều nơi như service khác, batch jobs...
Tăng tính tập trung (centralization)	Mọi logic liên quan đến voucher được gom lại, dễ hiểu hơn khi đọc code.

Bảng 2. 14 Lý do áp dụng Facade Pattern

Tóm lại: Áp dụng Facade Pattern để gom các xử lý voucher nhà hàng và hệ thống vào một interface đơn giản VoucherFacade, giúp controller không cần quan tâm đến chi tiết của từng service bên dưới và giảm thiểu sự phụ thuộc.

2.8.2 Sơ đồ lớp



Hình 2. 44 Sơ đồ lớp áp dụng Facade Pattern vào Voucher

2.8.3 Code áp dụng



```

@Service
public class ServiceVoucherRestaurant {

    private final DatabaseConnector dbConnector;
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    CallableStatement callableStatement = null;
    PreparedStatement updateStatement = null;

    PreparedStatement preparedStatement = null;

    public ServiceVoucherRestaurant() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database, username, password
        );
    }

    public VoucherRestaurant addVoucher(String name, int quantity, Double price, LocalDateTime expiry,
                                         int id_fk_restaurant) {
        int result = 0;
        try {
            connection = dbConnector.connect();
            String query = "INSERT INTO voucher_restaurant(Name,Quantity,Price,Expiry,Id_fk,Active)"
                           "VALUES (?, ?, ?, ?, ?, ?)";
            statement = connection.prepareStatement(query);
            statement.setString(1, name);
            statement.setInt(2, quantity);
            statement.setDouble(3, price);
            statement.setTimestamp(4, Timestamp.valueOf(expiry));
            statement.setInt(5, id_fk_restaurant);
            result = statement.executeUpdate();

            if (result > 0) {
                return new VoucherRestaurant(name, quantity, price, expiry, id_fk_restaurant, 0);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    public int active(int id) {
        try {
            connection = dbConnector.connect();
            String updateQuery = "UPDATE voucher_restaurant SET Active = 1 WHERE Id = ?";
            updateStatement = connection.prepareStatement(updateQuery);
            updateStatement.setInt(1, id);
            int rowsAffected = updateStatement.executeUpdate();
            if (rowsAffected > 0) {
                return 1;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }
    ...
}

```

Hình 2. 45 Lớp ServiceVoucherRestaurant

```

@Service
public class ServiceVoucherSystem {
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    CallableStatement callableStatement = null;
    PreparedStatement updateStatement = null;

    PreparedStatement preparedStatement = null;

    public ServiceVoucherSystem() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database, username, password
        );
    }

    public VoucherSystem addVoucher(String name, int quantity, Double price, LocalDateTime expiry, int id_fk) {
        int result = 0;
        try {
            connection = dbConnector.connect();
            String query = "INSERT INTO voucher_system(Name,Quantity,Price,Expiry,Id_fk) VALUES (?, ?, ?, ?, ?)";
            statement = connection.prepareStatement(query);
            statement.setString(1, name);
            statement.setInt(2, quantity);
            statement.setDouble(3, price);
            statement.setTimestamp(4, Timestamp.valueOf(expiry));
            statement.setInt(5, id_fk);

            result = statement.executeUpdate();

            if (result > 0) {
                return new VoucherSystem(name, quantity, price, expiry, id_fk);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

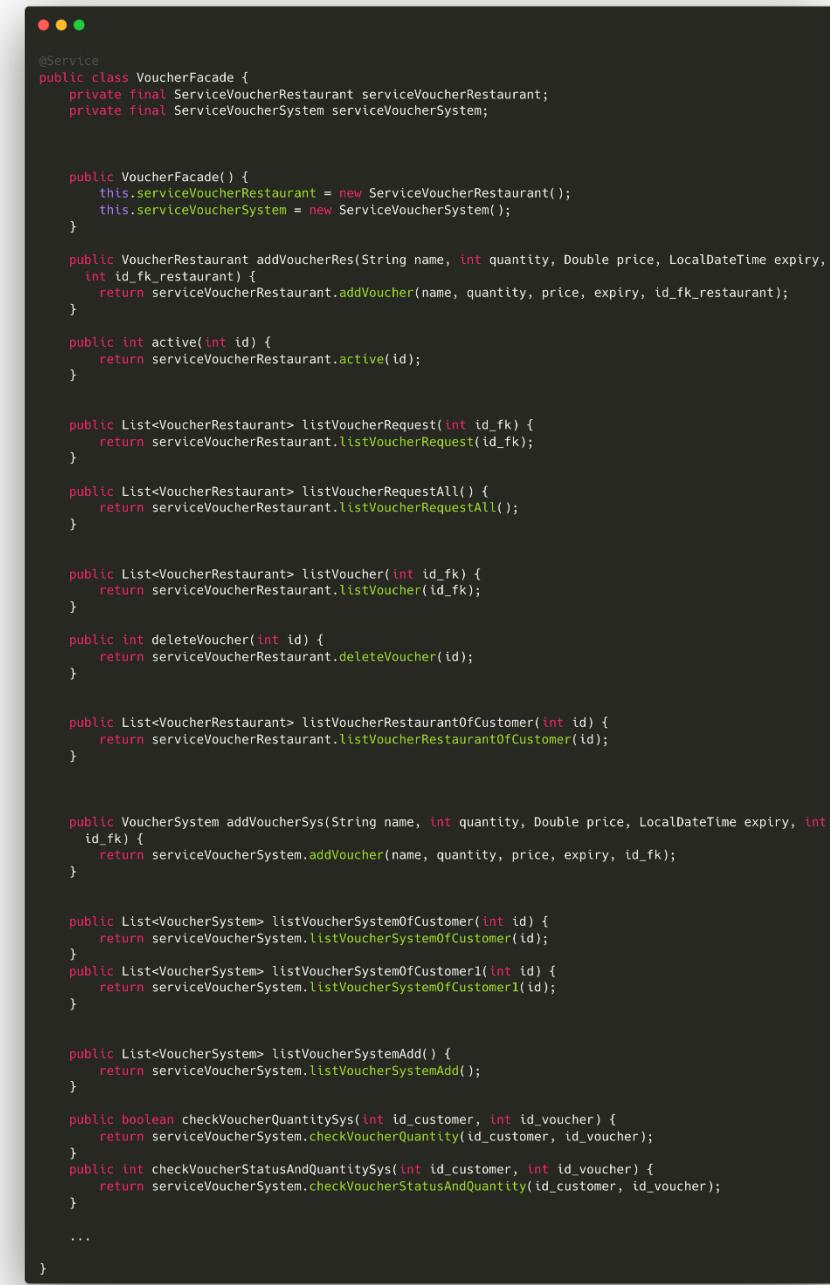
    public List<VoucherSystem> listVoucherSystemOfCustomer(int id) {
        List<VoucherSystem> voucherSystems = new ArrayList<>();
        try {
            connection = dbConnector.connect();
            String query = "SELECT r.* from get_voucher_system g, voucher_system r WHERE g.Id_customer = ? and g.Id_voucher_system = r.Id";
            statement = connection.prepareStatement(query);
            statement.setInt(1, id);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                VoucherSystem voucherSystem = new VoucherSystem();
                voucherSystem.setId(resultSet.getInt("Id"));
                voucherSystem.setName(resultSet.getString("Name"));
                voucherSystem.setPrice(resultSet.getDouble("Price"));
                voucherSystem.setQuantity(resultSet.getInt("Quantity"));
                Timestamp expiryTimestamp = resultSet.getTimestamp("Expiry");
                if (expiryTimestamp != null) {
                    LocalDateTime expiryDateTime = expiryTimestamp.toLocalDateTime();
                    voucherSystem.setExpiry(expiryDateTime);
                }
                voucherSystem.setId_system(resultSet.getInt("Id_fk"));
                voucherSystems.add(voucherSystem);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return voucherSystems;
    }

    public List<VoucherSystem> listVoucherSystemAdd() {
        List<VoucherSystem> voucherRestaurants = new ArrayList<>();
        try {
            connection = dbConnector.connect();
            String query = "SELECT * FROM voucher_system WHERE Expiry > NOW() And Quantity > 0";
            statement = connection.prepareStatement(query);
            resultSet = statement.executeQuery();

            while (resultSet.next()) {
                VoucherSystem voucherRestaurant = new VoucherSystem();
                voucherRestaurant.setId(resultSet.getInt("Id"));
                voucherRestaurant.setName(resultSet.getString("Name"));
                voucherRestaurant.setPrice(resultSet.getDouble("Price"));
                voucherRestaurant.setQuantity(resultSet.getInt("Quantity"));
                Timestamp expiryTimestamp = resultSet.getTimestamp("Expiry");
                if (expiryTimestamp != null) {
                    LocalDateTime expiryDateTime = expiryTimestamp.toLocalDateTime();
                    voucherRestaurant.setExpiry(expiryDateTime);
                }
                voucherRestaurant.setId_system(resultSet.getInt("Id_fk"));
                voucherRestaurants.add(voucherRestaurant);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return voucherRestaurants;
    }
}

```

Hình 2. 46 Lớp ServiceVoucherSystem



```

@Service
public class VoucherFacade {
    private final ServiceVoucherRestaurant serviceVoucherRestaurant;
    private final ServiceVoucherSystem serviceVoucherSystem;

    public VoucherFacade() {
        this.serviceVoucherRestaurant = new ServiceVoucherRestaurant();
        this.serviceVoucherSystem = new ServiceVoucherSystem();
    }

    public VoucherRestaurant addVoucherRes(String name, int quantity, Double price, LocalDateTime expiry,
                                           int id_fk_restaurant) {
        return serviceVoucherRestaurant.addVoucher(name, quantity, price, expiry, id_fk_restaurant);
    }

    public int active(int id) {
        return serviceVoucherRestaurant.active(id);
    }

    public List<VoucherRestaurant> listVoucherRequest(int id_fk) {
        return serviceVoucherRestaurant.listVoucherRequest(id_fk);
    }

    public List<VoucherRestaurant> listVoucherRequestAll() {
        return serviceVoucherRestaurant.listVoucherRequestAll();
    }

    public List<VoucherRestaurant> listVoucher(int id_fk) {
        return serviceVoucherRestaurant.listVoucher(id_fk);
    }

    public int deleteVoucher(int id) {
        return serviceVoucherRestaurant.deleteVoucher(id);
    }

    public List<VoucherRestaurant> listVoucherRestaurantOfCustomer(int id) {
        return serviceVoucherRestaurant.listVoucherRestaurantOfCustomer(id);
    }

    public VoucherSystem addVoucherSys(String name, int quantity, Double price, LocalDateTime expiry, int
                                         id_fk) {
        return serviceVoucherSystem.addVoucher(name, quantity, price, expiry, id_fk);
    }

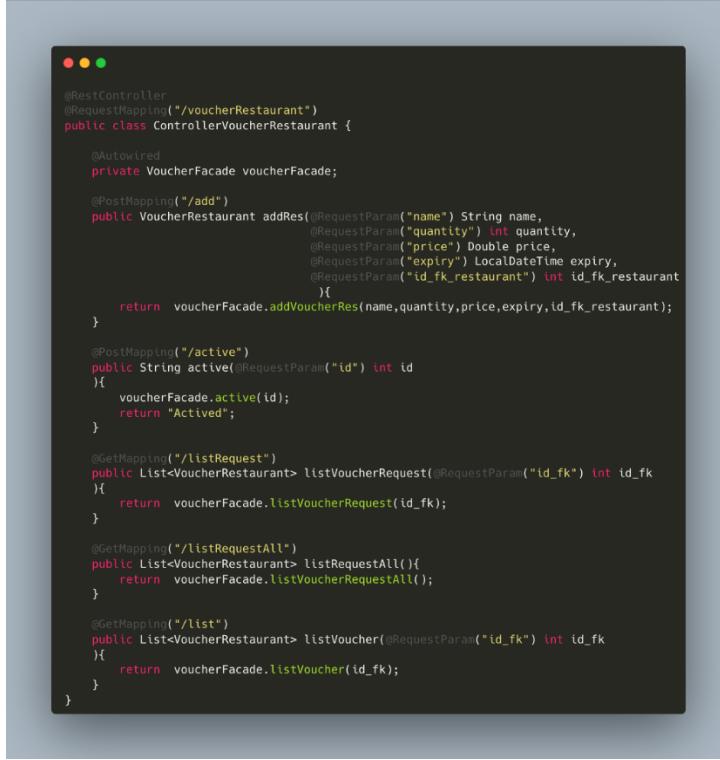
    public List<VoucherSystem> listVoucherSystemOfCustomer(int id) {
        return serviceVoucherSystem.listVoucherSystemOfCustomer(id);
    }
    public List<VoucherSystem> listVoucherSystemOfCustomer1(int id) {
        return serviceVoucherSystem.listVoucherSystemOfCustomer1(id);
    }

    public List<VoucherSystem> listVoucherSystemAdd() {
        return serviceVoucherSystem.listVoucherSystemAdd();
    }

    public boolean checkVoucherQuantitySys(int id_customer, int id_voucher) {
        return serviceVoucherSystem.checkVoucherQuantity(id_customer, id_voucher);
    }
    public int checkVoucherStatusAndQuantitySys(int id_customer, int id_voucher) {
        return serviceVoucherSystem.checkVoucherStatusAndQuantity(id_customer, id_voucher);
    }
    ...
}

```

Hình 2. 47 Lớp VoucherFacade



```

@RestController
@RequestMapping("/voucherRestaurant")
public class ControllerVoucherRestaurant {

    @Autowired
    private VoucherFacade voucherFacade;

    @PostMapping("/add")
    public VoucherRestaurant addRes(@RequestParam("name") String name,
                                    @RequestParam("quantity") int quantity,
                                    @RequestParam("price") Double price,
                                    @RequestParam("expiry") LocalDateTime expiry,
                                    @RequestParam("id_fk_restaurant") int id_fk_restaurant)
    {
        return voucherFacade.addVoucherRes(name, quantity, price, expiry, id_fk_restaurant);
    }

    @PostMapping("/active")
    public String active(@RequestParam("id") int id)
    {
        voucherFacade.active(id);
        return "Actived";
    }

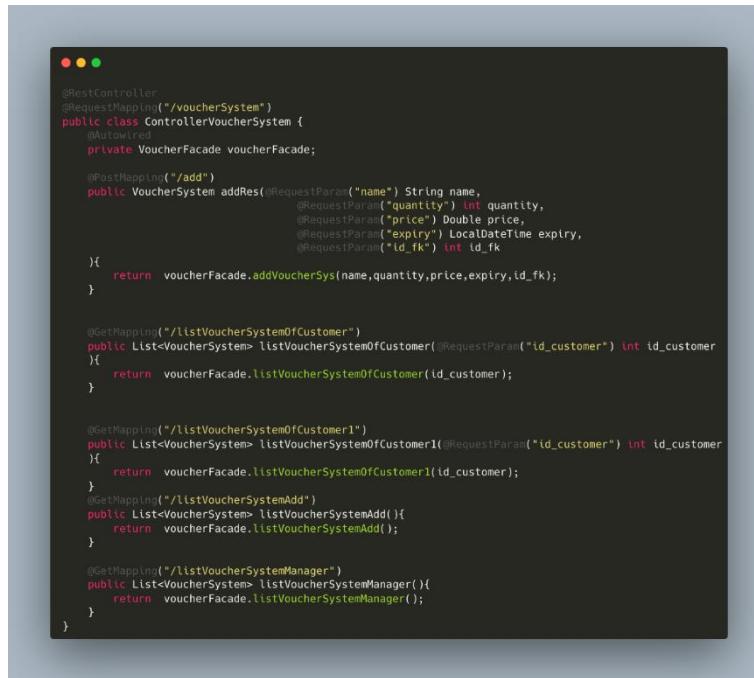
    @GetMapping("/listRequest")
    public List<VoucherRestaurant> listVoucherRequest(@RequestParam("id_fk") int id_fk)
    {
        return voucherFacade.listVoucherRequest(id_fk);
    }

    @GetMapping("/listRequestAll")
    public List<VoucherRestaurant> listRequestAll()
    {
        return voucherFacade.listVoucherRequestAll();
    }

    @GetMapping("/list")
    public List<VoucherRestaurant> listVoucher(@RequestParam("id_fk") int id_fk)
    {
        return voucherFacade.listVoucher(id_fk);
    }
}

```

Hình 2. 48 Lớp ControllerVoucherRestaurant



```

@RestController
@RequestMapping("/voucherSystem")
public class ControllerVoucherSystem {

    @Autowired
    private VoucherFacade voucherFacade;

    @PostMapping("/add")
    public VoucherSystem addRes(@RequestParam("name") String name,
                               @RequestParam("quantity") int quantity,
                               @RequestParam("price") Double price,
                               @RequestParam("expiry") LocalDateTime expiry,
                               @RequestParam("id_fk") int id_fk)
    {
        return voucherFacade.addVoucherSys(name, quantity, price, expiry, id_fk);
    }

    @GetMapping("/listVoucherSystemOfCustomer")
    public List<VoucherSystem> listVoucherSystemOfCustomer(@RequestParam("id_customer") int id_customer)
    {
        return voucherFacade.listVoucherSystemOfCustomer(id_customer);
    }

    @GetMapping("/listVoucherSystemOfCustomer1")
    public List<VoucherSystem> listVoucherSystemOfCustomer1(@RequestParam("id_customer") int id_customer)
    {
        return voucherFacade.listVoucherSystemOfCustomer1(id_customer);
    }

    @GetMapping("/listVoucherSystemAdd")
    public List<VoucherSystem> listVoucherSystemAdd()
    {
        return voucherFacade.listVoucherSystemAdd();
    }

    @GetMapping("/listVoucherSystemManager")
    public List<VoucherSystem> listVoucherSystemManager()
    {
        return voucherFacade.listVoucherSystemManager();
    }
}

```

Hình 2. 49 Lớp ControllerVoucherSystem

2.9 Proxy Pattern

2.9.1 Lý do áp dụng

2.9.1.1 Khái niệm

Proxy là một mẫu thiết kế (design pattern) thuộc nhóm cấu trúc (Structural Pattern), cho phép ta đại diện cho một đối tượng khác để kiểm soát quyền truy cập tới đối tượng đó.

Nói cách khác, Proxy là lớp trung gian nằm giữa client (người gọi) và real object (đối tượng thực thi). Nó giúp ta thêm các chức năng bổ sung như:

- Ghi log
- Kiểm tra phân quyền
- Giới hạn truy cập
- Tạo cache
- Trì hoãn khởi tạo (lazy loading)
- Giao tiếp mạng hoặc điều khiển từ xa (Remote Proxy)

2.9.1.2 Lý do áp dụng

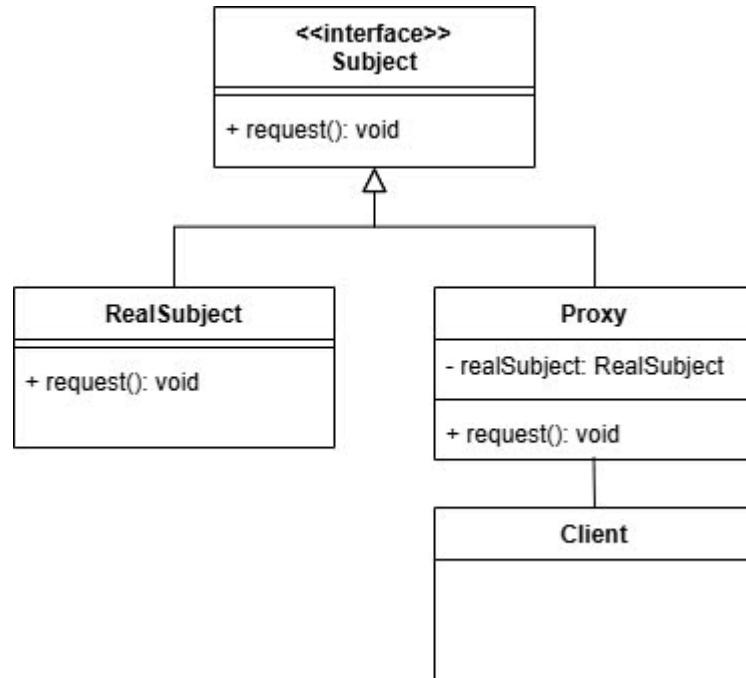
Lý do	Giải thích
Chức năng kiểm soát quyền truy cập	Proxy Pattern giúp kiểm soát và ghi lại mọi hành động của người dùng đối với các phương thức quan trọng như login, changePassword, verifyOTP, v.v. Trong ServiceUserProxy, mỗi phương thức đều có thêm việc ghi log thông tin về hành động (ví dụ: ghi log khi người dùng đăng nhập hoặc thay đổi mật khẩu). Điều này giúp quản lý, giám sát, và kiểm tra các thao tác mà người dùng thực hiện.

Tăng cường bảo mật và xử lý lỗi	Proxy Pattern có thể được sử dụng để thêm logic bảo mật như kiểm tra và ghi lại thông tin trước khi thực hiện hành động chính. Ví dụ, trong changePasswordReal, Proxy ghi lại thông tin và kiểm tra kết quả của hành động, giúp dễ dàng phát hiện lỗi và tăng cường bảo mật khi có sự cố.
Tách biệt giữa lớp thực thi và lớp giao diện	Proxy giúp tách biệt logic chính của ServiceUser khỏi các yêu cầu bổ sung (như ghi log, kiểm tra quyền truy cập, v.v.). Điều này giúp code dễ bảo trì và mở rộng, vì các thay đổi trong logic kiểm tra và bảo mật có thể được thực hiện trong ServiceUserProxy mà không cần thay đổi mã trong ServiceUser.
Dễ dàng thay thế hoặc mở rộng dịch vụ	Sử dụng Proxy giúp dễ dàng thay thế realServiceUser bằng một dịch vụ khác mà không ảnh hưởng đến các phần còn lại của hệ thống. Nếu cần thay đổi cách thức xử lý, ta chỉ cần thay đổi logic trong Proxy mà không làm gián đoạn hoạt động của hệ thống chính.
Giảm tải cho các yêu cầu không cần thiết	Proxy có thể xử lý các yêu cầu trước khi chuyển tiếp đến ServiceUser, giúp giảm bớt các thao tác không cần thiết, ví dụ như kiểm tra OTP hay đăng nhập. Thay vì trực

	tiếp gọi đến ServiceUser, Proxy có thể thực hiện các bước kiểm tra và chỉ chuyển tiếp khi thực sự cần thiết.
Hỗ trợ ghi lại thông tin và debug	Việc ghi lại log trong Proxy giúp người phát triển dễ dàng theo dõi quá trình thực thi và nhanh chóng phát hiện vấn đề. Việc này đặc biệt hữu ích trong việc debug, vì tất cả các hành động quan trọng đều có thông tin nhật ký, như khi người dùng đăng nhập hoặc thay đổi mật khẩu.

Bảng 2. 15 Lý do áp dụng Proxy Pattern

2.9.1.3 Phân tích vai trò các lớp



Hình 2. 50 Sơ đồ tổng quát của Proxy Pattern

Thành phần của lớp tổng quát

Cụ thể

Subject	IServiceUser
RealSubject	ServiceUser
Proxy	ServiceUserProxy
Client	ControllerRegister

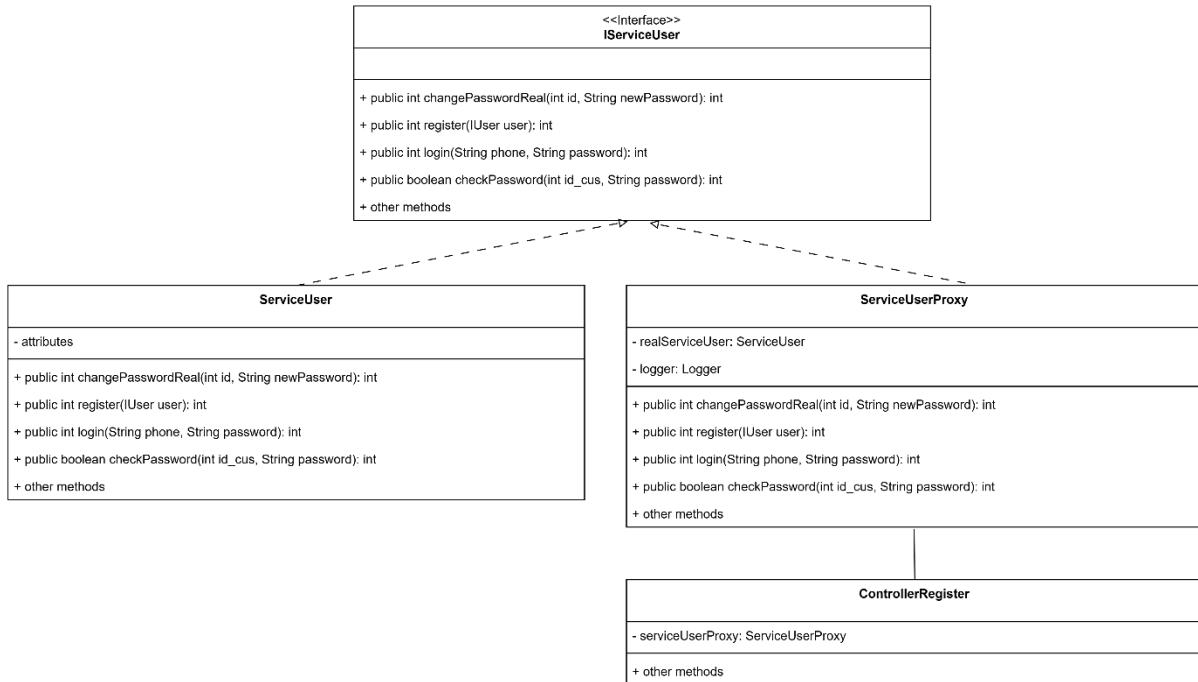
Bảng 2. 16 Các thành phần của Proxy Pattern

Tổng quan về các lớp:

- ServiceUserProxy: Là lớp trung gian (proxy) giúp ghi log, xử lý bảo mật, kiểm tra quyền truy cập và các hoạt động bổ sung trước khi chuyển tiếp yêu cầu đến ServiceUser. Vai trò chủ yếu của lớp này là kiểm soát luồng dữ liệu và bảo vệ dịch vụ thực thi chính.
- ServiceUser: Là lớp thực thi chính chịu trách nhiệm xử lý các nghiệp vụ cốt lõi liên quan đến người dùng như đăng nhập, đăng ký, đổi mật khẩu, kiểm tra OTP, v.v. Đây là nơi chứa logic thực sự cho các thao tác người dùng.
- ControllerRegister: Là lớp controller giúp nhận và xử lý các yêu cầu HTTP từ phía người dùng, như đăng ký, đăng nhập, thay đổi mật khẩu, xác thực OTP. Controller này tương tác với lớp ServiceUserProxy để thực hiện các thao tác nghiệp vụ.
- IServiceUser: Interface này định nghĩa các phương thức cần thiết mà ServiceUser và ServiceUserProxy phải triển khai, giúp đảm bảo tính đồng nhất trong việc thực hiện các thao tác người dùng và dễ dàng thay thế các lớp thực thi.
- User: Là lớp mô hình chứa các thuộc tính của người dùng, được sử dụng khi đăng ký và lưu trữ thông tin người dùng.

- IUser: Interface định nghĩa các phương thức mà lớp User phải triển khai, giúp hệ thống làm việc với đối tượng người dùng một cách linh hoạt và nhất quán.

2.9.2 Sơ đồ lớp



Hình 2. 51 Sơ đồ lớp áp dụng Proxy Pattern vào Service User

2.9.3 Code áp dụng

```
package com.example.Api.inheritance;

import java.util.Map;

public interface IServiceUser {
    public int changePasswordReal(int id, String newPassword);
    public int register(IUser user);
    public int login(String phone, String password);
    public boolean checkPassword(int id_cus, String
password);
    public int verifyOTP(String phone, String otp);
    public int resendOTP(String phone, String newOtp);
    public String getUserNameByPhone(String phone);
    public String getGenderByPhone(String phone);
    public int getUserRoleByPhone(String phone);
    public int getUserIdByPhone(String phone);
    public Map<String, Object> getUserByPhone(String phone);
    public int loginEmail(String email, String fullname);
    public int findUserByEmail(String email);
    public int findUserByPhone(String phone);
    public int changePassword(String phone, String password);
    public int getIdByEmail(String email);
}
```

Hình 2. 52 Lớp IServiceUser

```

● ● ●

@Service
public class ServiceUser implements IServicelUser {
    private final UserRoleFactory factory = new UserRoleFactory();
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    PreparedStatement selectStatement = null;
    PreparedStatement loginStatement = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    CallableStatement callableStatement = null;
    PreparedStatement updateStatement = null;
    PreparedStatement preparedStatement = null;
    PreparedStatement checkStatement = null;

    public ServiceUser() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database, username, password
        );
    }

    public int register(IUser user) {
        try {
            connection = dbConnector.connect();
            String checkQuery = "SELECT COUNT(*) FROM user WHERE PhoneNumber = ? and Active = 0";
            checkStatement = connection.prepareStatement(checkQuery);
            checkStatement.setString(1, user.getPhoneNumber());
            ResultSet resultSet = checkStatement.executeQuery();
            if (resultSet.next() && resultSet.getInt(1) > 0) {

                String updateQuery = "UPDATE user SET FullName = ?, PassW = ?, Otp = ?, Id_role = ? WHERE
PhoneNumber = ? and Active = 0";
                updateStatement = connection.prepareStatement(updateQuery);
                updateStatement.setString(1, user.getFullName());
                updateStatement.setString(2, user.getPassword());
                updateStatement.setString(3, user.getOtp());
                updateStatement.setInt(4, user.getId_fk_role());
                updateStatement.setString(5, user.getPhoneNumber());

                updateStatement.executeUpdate();
            } else {
                String storedProc = "{CALL Signup(?, ?, ?, ?, ?, ?)}";
                callableStatement = connection.prepareCall(storedProc);
                callableStatement.setString(1, user.getFullName());
                callableStatement.setString(2, user.getPhoneNumber());
                callableStatement.setString(3, user.getUsername());
                callableStatement.setString(4, user.getPassword());
                callableStatement.setInt(5, user.getId_fk_role());
                callableStatement.setString(6, user.getOtp());
                callableStatement.executeUpdate();
            }
            return 1;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }

    public int login(String phone, String password) {
        try {
            connection = dbConnector.connect();
            String loginQuery = "SELECT COUNT(*) FROM user WHERE PhoneNumber = ? AND PassW = ? AND Active
= 1";
            loginStatement = connection.prepareStatement(loginQuery);
            loginStatement.setString(1, phone);
            loginStatement.setString(2, password);

            resultSet = loginStatement.executeQuery();
            if (resultSet.next() && resultSet.getInt(1) > 0) {
                return 1;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }

    // Other methods
}

```

Hình 2. 53 Lớp ServiceUser

```

package com.example.Api.pattern.proxy;

import com.example.Api.inheritance.IServiceUser;
import com.example.Api.service.ServiceUser;
import com.example.Api.inheritance.IUser;
import org.springframework.stereotype.Service;
import java.util.Map;
import java.util.logging.Logger;

@Service
public class ServiceUserProxy implements IServiceUser {
    private final ServiceUser realServiceUser;
    private static final Logger logger = Logger.getLogger(ServiceUserProxy.class.getName());

    public ServiceUserProxy(ServiceUser realServiceUser) {
        this.realServiceUser = realServiceUser;
    }

    @Override
    public int changePasswordReal(int id, String newPassword) {
        logger.info("User with ID " + id + " is attempting to change their password.");
        int result = realServiceUser.changePasswordReal(id, newPassword);
        if (result == 1) {
            logger.info("User with ID " + id + " successfully changed their password.");
        } else {
            logger.warning("Failed to change password for user ID " + id);
        }
        return result;
    }

    @Override
    public int register(IUser user) {
        logger.info("User registration attempted for phone number: " + user.getPhoneNumber());
        int result = realServiceUser.register(user);
        if (result == 1) {
            logger.info("User registered successfully: " + user.getPhoneNumber());
        } else {
            logger.warning("User registration failed for: " + user.getPhoneNumber());
        }
        return result;
    }

    @Override
    public int login(String phone, String password) {
        logger.info("Login attempt for phone number: " + phone);
        int result = realServiceUser.login(phone, password);
        if (result == 1) {
            logger.info("User logged in successfully: " + phone);
        } else {
            logger.warning("Login failed for phone number: " + phone);
        }
        return result;
    }

    @Override
    public boolean checkPassword(int id_cus, String password) {
        logger.info("Checking password for user ID: " + id_cus);
        return realServiceUser.checkPassword(id_cus, password);
    }

    @Override
    public int verifyOTP(String phone, String otp) {
        logger.info("Verifying OTP for phone number: " + phone);
        return realServiceUser.verifyOTP(phone, otp);
    }

    @Override
    public int resendOTP(String phone, String newOtp) {
        logger.info("Resending OTP for phone number: " + phone);
        return realServiceUser.resendOTP(phone, newOtp);
    }

    // Other methods
}

```

Hình 2. 54 Lớp ServiceUserProxy



```

package com.example.Api.controller;

import com.example.Api.inheritance.IUser;
import com.example.Api.model.User;
import com.example.Api.pattern.proxy.ServiceUserProxy;
import com.example.Api.service.ServiceUser;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/register")
public class ControllerRegister {
    @Autowired
    ServiceUserProxy serviceUserProxy;

    public ControllerRegister(ServiceUser serviceUser) {
        this.serviceUserProxy = new ServiceUserProxy(serviceUser);
    }

    @GetMapping("/checkPassword")
    public Boolean checkPassword(@RequestParam("id_customer") int id_customer, @RequestParam("password") String password) {
        return serviceUserProxy.checkPassword(id_customer, password);
    }

    @PostMapping("/changePassword")
    public int changePassword(@RequestParam("id_customer") int id_customer, @RequestParam("password") String password) {
        return serviceUserProxy.changePasswordReal(id_customer, password);
    }

    @PostMapping("/verify-otp")
    public ResponseEntity<String> verifyOTP(@RequestParam("phone") String phone,
                                              @RequestParam("otp") String otp) {
        int result = serviceUserProxy.verifyOTP(phone, otp);
        if (result == 1) {
            return ResponseEntity.ok("OTP verified successfully.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Invalid OTP or phone number.");
        }
    }

    @PostMapping("/resend-otp")
    public ResponseEntity<String> resendOTP(@RequestParam("phone") String phone,
                                             @RequestParam("otp") String newotp) {
        int result = serviceUserProxy.resendOTP(phone, newotp);
        if (result == 1) {
            return ResponseEntity.ok("OTP verified successfully.");
        } else {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Invalid OTP or phone number.");
        }
    }

    // Other methods
}

```

Hình 2. 55 Lớp ControllerRegister

2.10 Visitor Pattern

2.10.1 Lý do áp dụng

2.10.1.1 Khái niệm

Visitor Pattern là một design pattern thuộc nhóm hành vi (behavioral), cho phép ta tách rời logic xử lý khỏi các đối tượng mà logic đó áp dụng lên. Thay vì mỗi lớp tự xử lý logic riêng, ta đưa logic đó vào một đối tượng "Visitor", rồi gửi đối tượng đó đến từng lớp để nó "được thăm".

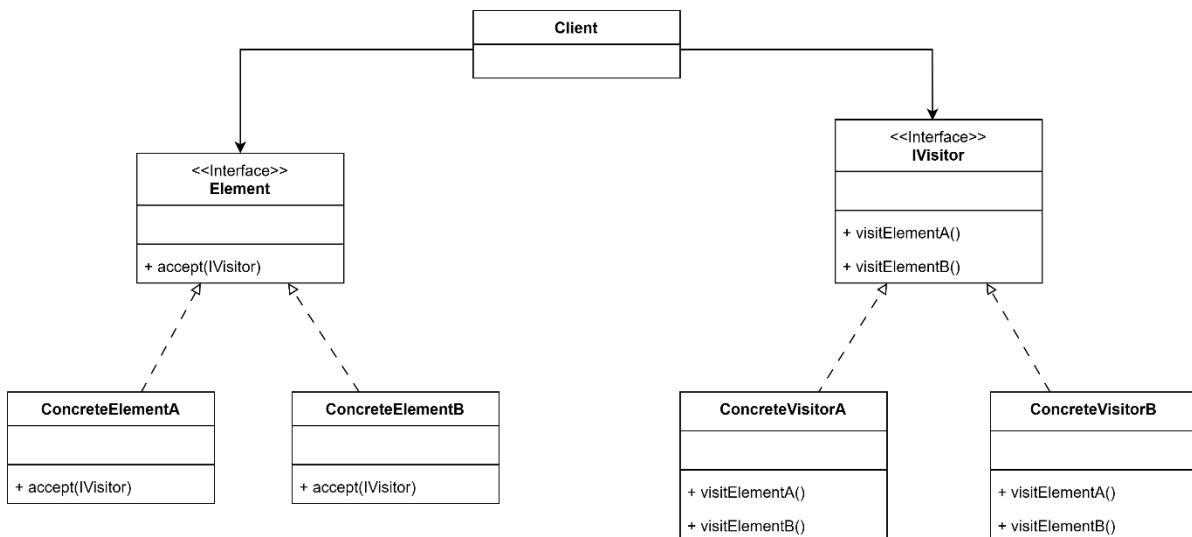
2.10.1.2 Lý do áp dụng

Bảng 2. 17 Lý do áp dụng Visitor Pattern

Lý do	Phân tích
Tách biệt logic xử lý khỏi cấu trúc dữ liệu	Hệ thống có các entity User, Shipper, Restaurant cùng kế thừa IUser. Thay vì nhét logic xử lý (ví dụ: thống kê, xuất file) vào từng class, ta tách riêng ra visitor.
Dễ mở rộng hành vi mà không sửa đổi lớp gốc	Khi cần thêm chức năng như in báo cáo, gửi email, lọc theo điều kiện..., chỉ cần tạo Visitor mới implement IUserVisitor, không cần sửa User, Shipper, Restaurant.
Áp dụng tốt với hệ thống có nhiều lớp con khác nhau	IUser có nhiều class con, mỗi class có cách xử lý khác nhau – ví dụ: thống kê đơn hàng chỉ áp dụng cho Restaurant, còn Shipper thì cần thống kê quãng đường đi được.
Tránh dùng instanceof hoặc if-else lồng nhau	Nhờ accept(visitor), hệ thống gọi đúng hàm xử lý (visitUser, visitShipper,

	visitRestaurant) mà không cần check thủ công từng loại bằng instanceof.
Dễ test và tái sử dụng logic xử lý	Logic trong Visitor riêng biệt, dễ mock hoặc test riêng biệt từng loại xử lý mà không cần tạo thực thể phức tạp của User hay Restaurant.

2.10.1.3 Phân tích vai trò các lớp



Hình 2. 56 Sơ đồ lớp tổng quát Visitor Pattern

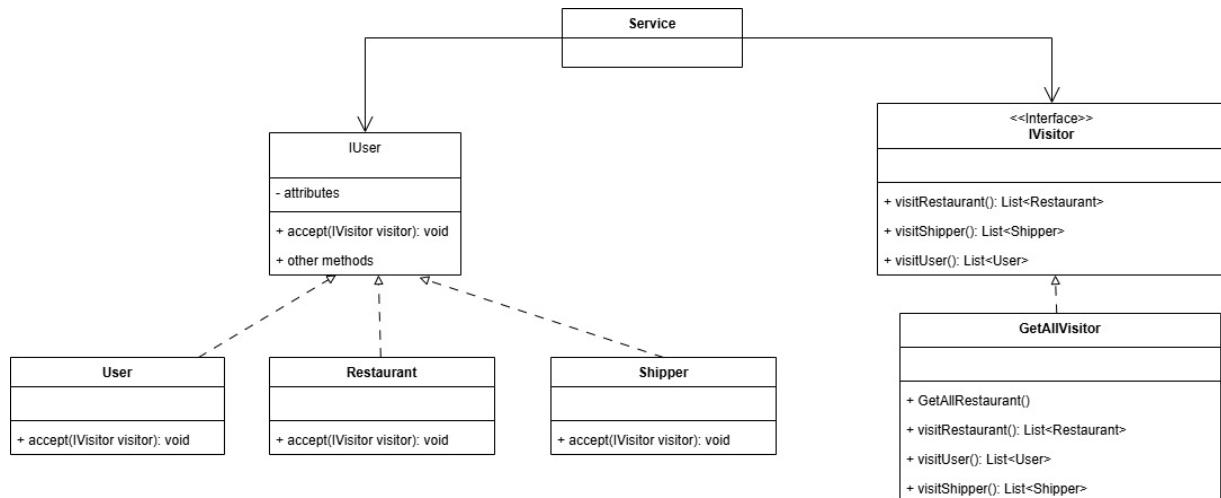
Thành phần của lớp tổng quát	Cụ thể
Client	RestaurantService
Element	IUser
IVisitor	IVisitor
ConcreteElement	User, Shipper, Restaurant
ConcreteVisitor	GetAllVisitor

Bảng 2. 18 Các thành phần của Visitor Pattern

Tổng quan như sau:

- IUser: Interface cha cho tất cả loại người dùng (User, Shipper, Restaurant)
- User, Shipper, Restaurant: Các class cụ thể, đại diện cho từng loại người dùng, cài đặt accept(visitor)
- IVisitor: Interface định nghĩa các hàm xử lý cho từng loại người dùng
- ConcreteVisitor (Ví dụ GetAllVisitor): Class hiện thực IVisitor, chứa logic xử lý riêng cho từng loại
- accept(visitor): Hàm trong mỗi IUser để gọi đúng phương thức tương ứng trong Visitor

2.10.2 Sơ đồ lớp



Hình 2. 57 Sơ đồ lớp áp dụng Visitor Pattern vào IUser

2.10.3 Code áp dụng

```
● ● ●

package com.example.Api.inheritance;

import com.example.Api.model.Restaurant;
import com.example.Api.model.Shipper;
import com.example.Api.model.User;

import java.util.List;

public interface IVisitor {
    public List<Restaurant> visitRestaurant();
    public List<User> visitUser();
    public List<Shipper> visitShipper();
}
```

Hình 2. 58 Lớp IVisitor

```
● ● ●

package com.example.Api.inheritance;

import java.util.Date;

public abstract class IUser {
    // Attributes

    public IUser() {
    }

    // Other constructors

    public abstract void accept(IVisitor visitor);

    // Other methods
}
```

Hình 2. 59 Lớp IUser

```
package com.example.Api.model;

import com.example.Api.inheritance.IUser;
import com.example.Api.inheritance.IVisitor;

public class Shipper extends IUser {
    private double balance;
    private int working;
    private double rating;
    private String bienso;
    private int status_request;
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public int getWorking() {
        return working;
    }

    public void setWorking(int working) {
        this.working = working;
    }

    public double getRating() {
        return rating;
    }

    public void setRating(double rating) {
        this.rating = rating;
    }

    public String getBienso() {
        return bienso;
    }

    public void setBienso(String bienso) {
        this.bienso = bienso;
    }

    public int getStatus_request() {
        return status_request;
    }

    public void setStatus_request(int status_request)
    {
        this.status_request = status_request;
    }

    public void accept(IVisitor visitor) {
        visitor.visitShipper();
    }
}
```

Hình 2. 60 Lớp Shipper

```
● ● ●

package com.example.Api.model;

import com.example.Api.inheritance.IUser;
import com.example.Api.inheritance.IVisitor;

public class Restaurant extends IUser {
    private int id;
    private String name;
    private Double balance;
    private int working;
    private Double rating;
    private String image;
    private Double latitude;
    private Double longitude;
    private Double distance;

    public Restaurant() {
    }

    public Restaurant(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public Restaurant(int id, String name, Double balance, int working, Double rating)
    {
        this.id = id;
        this.name = name;
        this.balance = balance;
        this.working = working;
        this.rating = rating;
    }

    public void accept(IVisitor visitor) {
        visitor.visitRestaurant();
    }

    // Other methods
}
```

Hình 2. 61 Lớp Restaurant

```
package com.example.Api.model;

import com.example.Api.inheritance.IUser;
import com.example.Api.inheritance.IVisitor;

import java.util.Date;

public class User extends IUser {
    public User() {
        super();
    }

    public User(String fullName, String phoneNumber, String username, String password, int id_fk_role) {
        super(fullName, phoneNumber, username, password, id_fk_role);
    }

    public User(String fullName, String phoneNumber, String username, String password, int id_fk_role,
String otp) {
        super(fullName, phoneNumber, username, password, id_fk_role, otp);
    }

    public User(int id, String fullName, String gender, Date birthday, String address, String
phoneNumber, String email, String username, String password, String otp, int active, int id_fk_role) {
        super(id, fullName, gender, birthday, address, phoneNumber, email, username, password, otp,
active, id_fk_role);
    }

    public void accept(IVisitor visitor) {
        visitor.visitUser();
    }
}
```

Hình 2. 62 Lớp User

```

package com.example.Api.pattern.visitor;

import com.example.Api.inheritance.IVisitor;
import com.example.Api.model.Restaurant;
import com.example.Api.model.Shipper;
import com.example.Api.model.User;
import com.example.Api.pattern.template.DatabaseConnector;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class GetAllVisitor implements IVisitor {
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    CallableStatement callableStatement = null;

    public GetAllVisitor(DatabaseConnector dbConnector) {
        this.dbConnector = dbConnector;
    }

    @Override
    public List<User> visitUser() {
        List<User> customers = new ArrayList<>();
        String query = "SELECT r.*, n.image, n.Address FROM customer r, user n WHERE r.id = n.Id";

        try {
            connection = dbConnector.connect();
            PreparedStatement statement = connection.prepareStatement(query);

            try (ResultSet resultSet = statement.executeQuery()) {
                while (resultSet.next()) {
                    User user = new User();
                    user.setId(resultSet.getInt("Id"));
                    user.setImage(resultSet.getString("image"));
                    user.setAddress(resultSet.getString("Address"));
                    customers.add(user);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return customers;
    }

    @Override
    public List<Shipper> visitShipper() {
        List<Shipper> shippers = new ArrayList<>();
        String query = "SELECT r.*, n.image, n.FullName, n.Address FROM shiper r, user n WHERE r.id = n.Id AND status_request = 0 or status_request = 1";

        try {
            connection = dbConnector.connect();
            PreparedStatement statement = connection.prepareStatement(query);

            try (ResultSet resultSet = statement.executeQuery()) {
                while (resultSet.next()) {
                    Shipper shipper = new Shipper();
                    shipper.setId(resultSet.getInt("Id"));
                    shipper.setName(resultSet.getString("FullName"));
                    shipper.setImage(resultSet.getString("image"));
                    shipper.setAddress(resultSet.getString("Address"));
                    shipper.setStatusRequest(resultSet.getInt("status_request"));
                    shippers.add(shipper);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return shippers;
    }

    @Override
    public List<Restaurant> visitRestaurant() {
        List<Restaurant> restaurants = new ArrayList<>();
        try {
            connection = dbConnector.connect();
            String query = "SELECT r.*, n.image, n.Address FROM restaurant r, user n where r.Id = n.Id";
            statement = connection.prepareStatement(query);
            resultSet = statement.executeQuery();
            while (resultSet.next()) {
                Restaurant restaurant = new Restaurant();
                restaurant.setId(resultSet.getInt("Id"));
                restaurant.setName(resultSet.getString("Name"));
                restaurant.setBalance(resultSet.getDouble("Balance"));
                restaurant.setWorking(resultSet.getInt("Working"));
                restaurant.setRating(resultSet.getDouble("Rating"));
                restaurant.setImage(resultSet.getString("image"));
                restaurant.setAddress(resultSet.getString("Address"));
                restaurants.add(restaurant);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return restaurants;
    }
}

```

Hình 2. 63 Lớp GetAllVisitor

```
● ● ●

package com.example.Api.service;

import com.example.Api.inheritance.IVisitor;
import com.example.Api.model.Restaurant;
import com.example.Api.pattern.template.DatabaseConnector;
import com.example.Api.pattern.visitor.GetAllVisitor;
import org.springframework.stereotype.Service;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

import static com.example.Api.global.Connection.*;
import static com.example.Api.global.Connection.password;

@Service
public class ServiceRestaurant {
    private final DatabaseConnector dbConnector;
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    CallableStatement callableStatement = null;
    private final IVisitor visitor;

    public ServiceRestaurant() {
        this.dbConnector = DatabaseConnector.getInstance(
            type_connection, database, username, password
        );
        this.visitor = new GetAllVisitor(dbConnector);
    }

    public List<Restaurant> listAllRestaurantAdmin() {
        return visitor.visitRestaurant();
    }

    // Other methods
}
```

Hình 2. 64 Lớp ServiceRestaurant

CHƯƠNG 3 – TỔNG KẾT

Sau quá trình đánh giá và phân tích, ứng dụng đặt và giao món ăn đã được cải thiện toàn diện nhằm khắc phục các vấn đề tồn tại trong phiên bản ban đầu như quy tắc đặt tên không nhất quán, cấu trúc mã lộn xộn, thiếu tính mở rộng và khó bảo trì. Việc tái cấu trúc được thực hiện dựa trên các nguyên lý thiết kế phần mềm hiện đại, giúp nâng cao chất lượng mã nguồn và hiệu suất hệ thống.

Các thành phần trong ứng dụng đã được phân tách rõ ràng theo từng vai trò và trách nhiệm, từ đó tăng cường khả năng mở rộng và dễ dàng tích hợp các tính năng mới trong tương lai. Ngoài ra, việc áp dụng các design pattern phù hợp như Strategy, Template Method, Observer, Visitor,... cũng góp phần cải thiện tính linh hoạt và tái sử dụng của mã nguồn.

Nhờ những cải tiến trên, ứng dụng không chỉ vận hành ổn định và hiệu quả hơn, mà còn mang đến trải nghiệm người dùng tốt hơn, đồng thời tạo nền tảng vững chắc để mở rộng quy mô và phát triển thêm các chức năng mới trong tương lai.

Tuy các cải tiến có thể còn tồn tại một số hạn chế và sai sót nhất định, nhưng đó là kết quả của quá trình nỗ lực không ngừng từ phía nhóm trong việc học hỏi, nghiên cứu và áp dụng các kiến thức đã được tiếp thu. Nhóm rất mong nhận được những nhận xét thẳng thắn và góp ý chân thành từ thầy để có thể tiếp tục hoàn thiện sản phẩm và phát triển hơn nữa trong các dự án sau này.

TÀI LIỆU THAM KHẢO

Tiếng Việt

1. Visitor Design Pattern - Trợ thủ đắc lực của Developers - <https://s.pro.vn/S2vi>.
Truy cập: 12/04/2025.
2. Proxy Design Pattern - Trợ thủ đắc lực của Developers - <https://s.pro.vn/DhMc>. Truy cập: 13/04/2025.
3. Các tài liệu môn học Design Pattern – Khoa CNTT Đại học Tôn Đức Trắng.

Tiếng Anh

1. Observer - <https://short.com.vn/kgRk>. Truy cập: 13/04/2025.
2. Design Patterns - <https://s.pro.vn/wca0>. Truy cập: 13/04/2025.
3. Visitor - <https://short.com.vn/01u6>. Truy cập: 29/03/2025.
4. Gangs of Four (GoF) Design Patterns - <https://s.pro.vn/2IQz>. Truy cập: 13/04/2025.
5. Proxy Design Pattern - <https://s.pro.vn/K5gy>. Truy cập: 13/04/2025.
6. Proxy - <https://short.com.vn/Uk9e>. Truy cập: 13/04/2025.
7. Design Patterns - Facade Pattern - <https://s.pro.vn/85YY>. Truy cập: 13/04/2025.