TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**Nguyễn Lê Quốc Trung – 52100860**

**Trương Anh Kiệt – 52100053**

**WEBSITE QUẢN LÝ NHÀ SÁCH**

**BÁO CÁO CUỐI KỲ**

**MẪU THIẾT KẾ**

**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**



**Nguyễn Lê Quốc Trung – 52100860**

**Trương Anh Kiệt – 52100053**

**WEBSITE QUẢN LÝ NHÀ SÁCH**

**BÁO CÁO CUỐI KỲ**

**MẪU THIẾT KẾ**

Người hướng dẫn

**GV. Vũ Đình Hồng**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025**

**LỜI CẢM ƠN**

Chúng em xin chân thành cảm ơn đến Trường Đại học Tôn Đức Thắng đã tạo điều kiện học tập tốt cho chúng em để có cơ hội tích lũy thêm kiến thức tại trường. Đồng thời chúng em cũng xin cảm ơn đến thầy Vũ Đình Hồng đã hỗ trợ giúp đỡ nhóm em thực hiện dự án cuối kì môn Mẫu thiết kế này.

*TP. Hồ Chí Minh, ngày 02 tháng 05 năm 2025*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*

*Trung*

*Nguyễn Lê Quốc Trung*

*Kiệt*

*Trương Anh Kiệt*

**CÔNG TRÌNH ĐƯỢC HOÀN THÀNH**

**TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi và được sự hướng dẫn khoa học của GV. 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 Dự 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 Dự 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 02 tháng 05 năm 2025*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*

*Trung*

*Nguyễn Lê Quốc Trung*

*Kiệt*

*Trương Anh Kiệt*

**WEBSITE QUẢN LÝ NHÀ SÁCH**

**TÓM TẮT**

Bài báo cáo cuối kì này bao gồm các design pattern trong dự án mà nhóm chúng em đã thực hiện áp dụng vào. Bao gồm các pattern là State, Command, Factory, Facade, Iterator, Singleton, Strategy, Observer và Template method pattern.

**MỤC LỤC**

[DANH MỤC HÌNH VẼ vii](#_Toc197296264)

[CHƯƠNG 1. MỞ ĐẦU VÀ TỔNG QUAN ĐỀ TÀI 1](#_Toc197296265)

[1.1 Lý do chọn đề tài 1](#_Toc197296266)

[1.2 Mục tiêu thực hiện đề tài 1](#_Toc197296267)

[CHƯƠNG 2. CƠ SỞ LÝ THUYẾT 2](#_Toc197296268)

[2.1 Singleton Pattern 2](#_Toc197296269)

[2.1.1 Cơ sở lý thuyết 2](#_Toc197296270)

[2.1.2 Lý do áp dụng 2](#_Toc197296271)

[2.1.3 Sơ đồ lớp 3](#_Toc197296272)

[2.1.4 Code áp dụng 3](#_Toc197296273)

[2.2 Strategy Pattern 4](#_Toc197296274)

[2.2.1 Cơ sở lý thuyết 4](#_Toc197296275)

[2.2.2 Lý do áp dụng 5](#_Toc197296276)

[2.2.3 Sơ đồ lớp 5](#_Toc197296277)

[2.2.4 Code áp dụng 5](#_Toc197296278)

[2.3 Template Method Pattern 6](#_Toc197296279)

[2.3.1 Cơ sở lý thuyết 6](#_Toc197296280)

[2.3.2 Lý do áp dụng 7](#_Toc197296281)

[2.3.3 Sơ đồ lớp 7](#_Toc197296282)

[2.3.4 Code áp dụng 8](#_Toc197296283)

[2.4 Iterator Pattern 10](#_Toc197296284)

[2.4.1 Cơ sở lý thuyết 10](#_Toc197296285)

[2.4.2 Lý do áp dụng 12](#_Toc197296286)

[2.4.3 Sơ đồ lớp 12](#_Toc197296287)

[2.4.4 Code áp dụng 12](#_Toc197296288)

[2.5 Observer Pattern 14](#_Toc197296289)

[2.5.1 Cơ sở lý thuyết 14](#_Toc197296290)

[2.5.2 Lý do áp dụng 15](#_Toc197296291)

[2.5.3 Sơ đồ lớp 17](#_Toc197296292)

[2.5.4 Code áp dụng 17](#_Toc197296293)

[2.6 State Pattern 19](#_Toc197296294)

[2.6.1 Cơ sở lý thuyết 19](#_Toc197296295)

[2.6.2 Lý do áp dụng 20](#_Toc197296296)

[2.6.3 Sơ đồ lớp 20](#_Toc197296297)

[2.6.4 Code áp dụng 21](#_Toc197296298)

[2.7 Command Pattern 23](#_Toc197296299)

[2.7.1 Cơ sở lý thuyết 23](#_Toc197296300)

[2.7.2 Lý do áp dụng 24](#_Toc197296301)

[2.7.4 Sơ đồ lớp 26](#_Toc197296302)

[2.7.5 Code áp dụng 26](#_Toc197296303)

[2.8 Facade Pattern 30](#_Toc197296304)

[2.8.1 Cơ sở lý thuyết 30](#_Toc197296305)

[2.8.2 Lý do áp dụng 32](#_Toc197296306)

[2.8.3 Sơ đồ lớp 33](#_Toc197296307)

[2.8.4 Code áp dụng 33](#_Toc197296308)

[2.9 Factory Pattern 36](#_Toc197296309)

[2.9.1 Cơ sở lý thuyết 36](#_Toc197296310)

[2.9.2 Lý do áp dụng 37](#_Toc197296311)

[2.9.3 Sơ đồ lớp 37](#_Toc197296312)

[2.9.4 Code áp dụng 38](#_Toc197296313)

[CHƯƠNG 3. KẾT LUẬN 42](#_Toc197296314)

[TÀI LIỆU THAM KHẢO 43](#_Toc197296315)

# MỞ ĐẦU VÀ TỔNG QUAN ĐỀ TÀI

## Lý do chọn đề tài

Hệ thống website quản lý nhà sách là hệ thống cung cấp các chức năng liên quan đến việc thao tác quản lý nhà sách. Đây là một hệ thống phức tạp nếu viết một cách thủ công. Từ đó việc áp dụng các design pattern góp phần giảm thiểu mức độ phức tạp của dự án cũng như tăng tính mở rộng, tái sử dụng và bảo trì hệ thống. Đó là lý do nhóm em quyết định áp dụng mẫu thiết kế vào dự án website quản lý nhà sách trên.

## Mục tiêu thực hiện đề tài

* Áp dụng hiệu quả các mẫu thiết kế vào bài dự án
* Tăng hiệu suất hệ thống
* Tăng khả năng tái sử dụng code
* Xây dựng website quản lý nhà sách đảm bảo các chức năng cơ bản

# CƠ SỞ LÝ THUYẾT

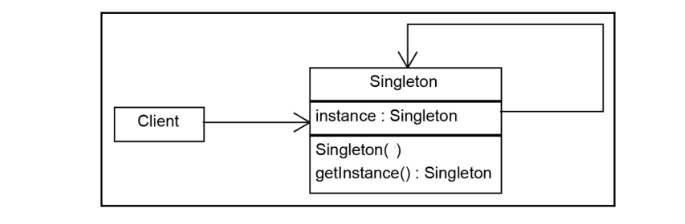
## Singleton Pattern

### Cơ sở lý thuyết

#### Khái niệm

Singleton Pattern là một mẫu thiết kế chỉ tạo ra duy nhất 1 một thể hiện (instance) của một class và cung cấp phương thức toàn cục để truy cập đến nó ở bất kỳ đâu.

#### Nội dung

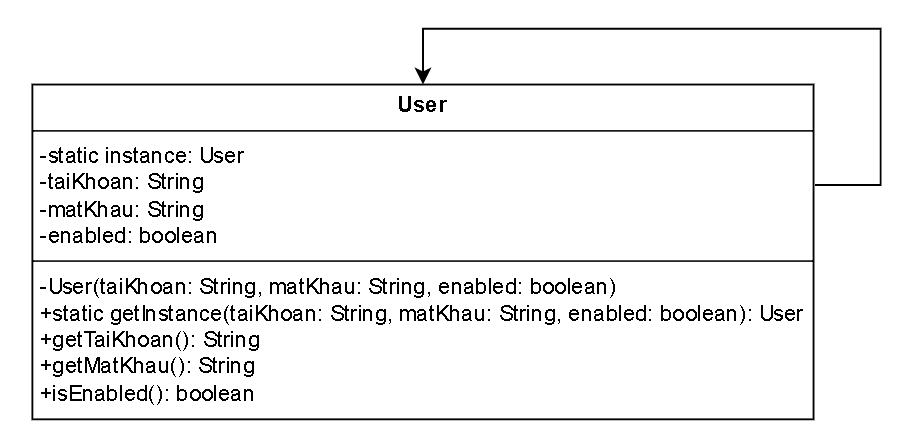


* Phương thức khởi tạo (constructor) được khai báo private để ngăn chặn việc tạo ra các instance mới của lớp từ bên ngoài
* Phương thức tĩnh (static) được cung cấp để truy cập instance duy nhất của lớp
* Instance được tạo ra và lưu trữ trong biến tĩnh static nội bộ của lớp, cung cấp một phương thức tĩnh getInstance() để lấy instance.

### Lý do áp dụng

Mẫu Singleton được áp dụng vào UserService để đảm bảo chỉ có một thể hiện duy nhất, giúp quản lý tập trung việc xác thực người dùng thông qua UserDetailsService và truy cập dữ liệu từ TaiKhoanRepository, đồng thời đảm bảo an toàn luồng với cơ chế double-checked locking, phù hợp cho các yêu cầu đồng thời trong ứng dụng web, và tối ưu hóa tài nguyên bằng cách tránh tạo nhiều đối tượng dịch vụ, dù Spring đã quản lý bean @Service như singleton mặc định.

### Sơ đồ lớp



### Code áp dụng

@Service  
public class UserService implements UserDetailsService {  
 private static UserService *instance*;  
 private final TaiKhoanRepository taiKhoanRepository;  
  
 private UserService(TaiKhoanRepository taiKhoanRepository) {  
 this.taiKhoanRepository = taiKhoanRepository;  
 }  
  
 public static UserService getInstance(TaiKhoanRepository repository) {  
 if (*instance* == null) {  
 synchronized (UserService.class) {  
 if (*instance* == null) {  
 *instance* = new UserService(repository);  
 }  
 }  
 }  
 return *instance*;  
 }  
  
 @Override  
 public UserDetails loadUserByUsername(String taiKhoanName) throws UsernameNotFoundException {  
 TaiKhoan foundTK = taiKhoanRepository.findById(taiKhoanName)  
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));  
 return new User(foundTK.getTaiKhoan(), foundTK.getMatKhau(), new ArrayList<>());  
 }  
}

Đoạn code trên định nghĩa lớp UserService, được đánh dấu bằng annotation @Service của Spring, triển khai giao diện UserDetailsService để hỗ trợ xác thực người dùng trong Spring Security. Lớp này áp dụng mẫu thiết kế Singleton bằng cách sử dụng biến tĩnh private instance để lưu trữ thể hiện duy nhất, hàm tạo private UserService (TaiKhoanRepository taiKhoanRepository) để ngăn khởi tạo từ bên ngoài, và phương thức tĩnh getInstance (TaiKhoanRepository repository) với cơ chế double-checked locking để đảm bảo an toàn luồng khi tạo thể hiện trong môi trường đa luồng. Phương thức loadUserByUsername thực thi yêu cầu của UserDetailsService, truy xuất thông tin người dùng từ TaiKhoanRepository và trả về đối tượng UserDetails để xử lý xác thực, ném ra ngoại lệ UsernameNotFoundException nếu không tìm thấy người dùng. Mặc dù Spring tự động quản lý bean @Service như singleton, việc triển khai Singleton thủ công có thể nhằm đảm bảo tính nhất quán hoặc sử dụng ngoài ngữ cảnh Spring, tuy nhiên, điều này có thể bị coi là dư thừa trong môi trường Spring..

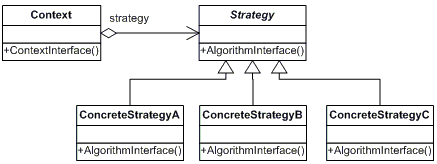
## Strategy Pattern

### Cơ sở lý thuyết

#### Khái niệm

Strategy Pattern cho phép xác định một tập hợp các thuật toán, đóng gói từng thuật toán và làm chúng có thể thay đổi tùy theo nhu cầu một cách độc lập với các client sử dụng nó.

#### Nội dung

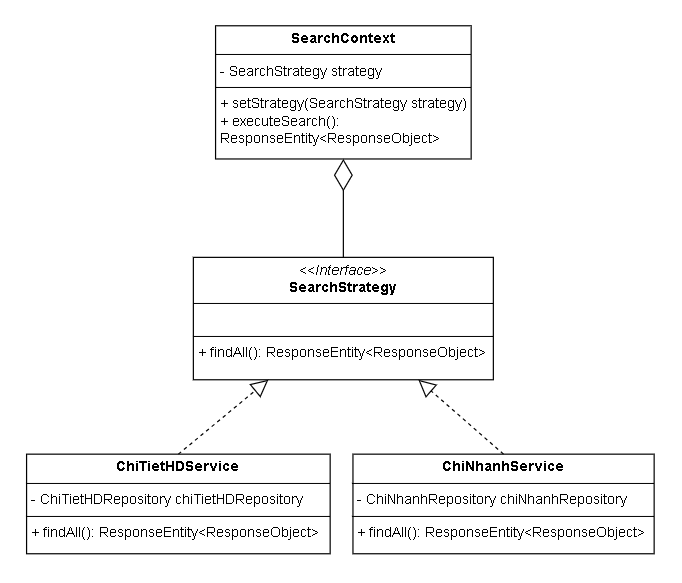


* Context: lớp thêm chiếu đến strategy object và gọi các phương thức của nó, có thể thay đổi cấu hình Strategy cho phù hợp với yêu cầu từ Client.
* Stategy: Interface chung cho context giao tiếp với stategy object.
* Concrete Strategy: Implement các thuật toán khác nhau cho Context sử dụng.

### Lý do áp dụng

Áp dụng strategy pattern vào việc thay đổi phương thức thanh toán trong lúc runtime của website tùy thuộc vào sự lựa chọn của người dùng mà không cần thay đổi mã nguồn cố định giúp cho việc mở rộng và bảo trì hệ thống trở lên dễ dàng hơn.

### Sơ đồ lớp



### Code áp dụng

// Interface định nghĩa chiến lược tìm kiếm  
interface SearchStrategy {  
 public ResponseEntity<ResponseObject> findAll();  
}

Tại ChiNhanhService:

@Service  
public class ChiNhanhService implements SearchStrategy{  
 @Autowired  
 private ChiNhanhRepository chiNhanhRepository;  
 @Autowired  
 private TaiKhoanRepository taiKhoanRepository;  
 @Autowired  
 private VerificationTokenRepository verificationTokenRepository;  
 @Autowired  
 private PasswordEncoder passwordEncoder;  
 @Autowired  
 private ApplicationEventPublisher publisher;  
 @Autowired  
 private AccountSessionScope accountSessionScope;  
  
 public ResponseEntity<ResponseObject> findAll() {  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Query All ChiNhanh Success", chiNhanhRepository.findAll())  
 );  
 }

Tại ChiTietHDService:

@Service  
public class ChiTietHDService implements SearchStrategy {  
 @Autowired  
 ChiTietHDRepository chiTietHDRepository;  
 @Autowired  
 ChiNhanhRepository chiNhanhRepository;  
 @Autowired  
 SanPhamRepository sanPhamRepository;  
 @Autowired  
 HoaDonRepository hoaDonRepository;  
 @Autowired  
 QuanLySPService quanLySPService;  
 @Autowired  
 private PhieuFactoryProvider phieuFactoryProvider;  
  
 public ResponseEntity<ResponseObject> findAll() {  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Query All ChiTietHD Success", chiTietHDRepository.findAll())  
 );  
 }

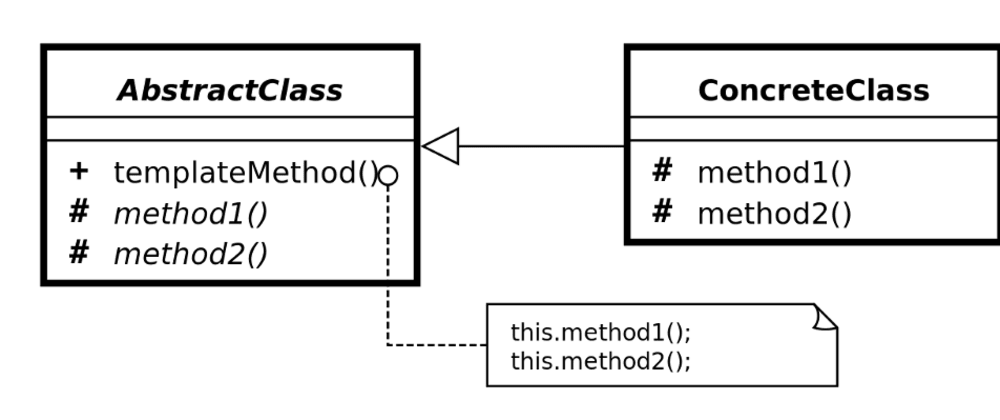
## Template Method Pattern

### Cơ sở lý thuyết

#### Khái niệm

Template Method Pattern được áp dụng vào deletePhieuNhap trong PhieuNhapService vì quy trình xóa phiếu nhập gồm nhiều bước cố định (kiểm tra phiếu, tìm chi tiết, cập nhật kho, xóa chi tiết, xóa phiếu) có thể tái sử dụng cho các loại phiếu khác như PhieuChuyenSanPham hay HoaDon. Pattern này chuẩn hóa quy trình trong AbstractPhieuDeletion, cho phép PhieuNhapDeletion triển khai các bước cụ thể, giảm lặp mã, tăng tính bảo trì và dễ mở rộng khi thêm loại phiếu mới, mà không làm hệ thống phức tạp hay gượng ép

#### Nội dung

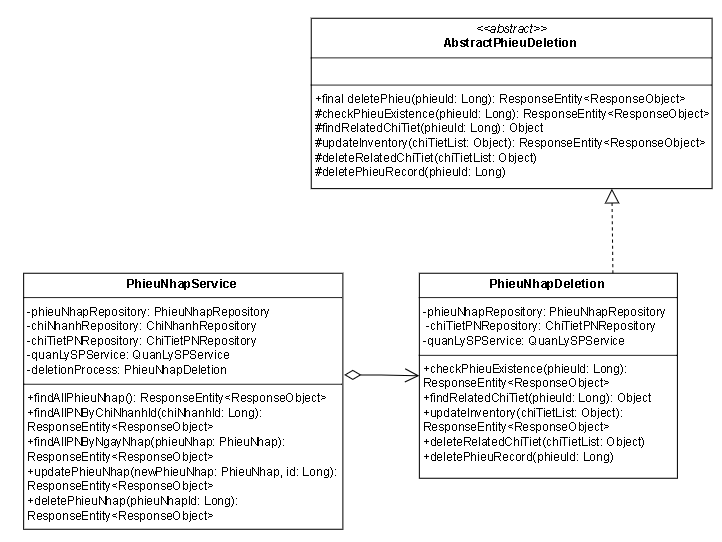


* Abtract Class: định nghĩa Template Method bao gồm các bước cơ bản của thuật toán.
* Concrete Class: cài đặt các bước cụ thể của thuật toán bằng các ghi đè các phương thức trong lớp cha.

### Lý do áp dụng

Áp dụng template method vào việc chuẩn bị các thông tin và gửi mail cho người dùng tùy thuộc vào các loại email như thông tin chuyến bay đã đặt, chuyến bay chậm trễ, hủy đơn đặt, … nhưng vẫn theo một quy trình nhất định, chỉ khác nội dung các loại email giúp cho việc triển khai trở nên dễ dàng và tránh trùng lặp code.

### Sơ đồ lớp



### Code áp dụng

public abstract class AbstractPhieuDeletion {  
 // Template method: Quy trình xóa phiếu  
 public final ResponseEntity<ResponseObject> deletePhieu(Long phieuId) {  
 // Bước 1: Kiểm tra sự tồn tại của phiếu  
 ResponseEntity<ResponseObject> checkResult = checkPhieuExistence(phieuId);  
 if (checkResult.getStatusCode() != HttpStatus.*OK*) {  
 return checkResult;  
 }  
  
 // Bước 2: Tìm danh sách chi tiết liên quan  
 Object chiTietList = findRelatedChiTiet(phieuId);  
 if (chiTietList == null) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "No related ChiTiet found for phieuId = " + phieuId, ""));  
 }  
  
 // Bước 3: Cập nhật số lượng kho  
 ResponseEntity<ResponseObject> updateResult = updateInventory(chiTietList);  
 if (updateResult != null && updateResult.getStatusCode() != HttpStatus.*OK*) {  
 return updateResult;  
 }  
  
 // Bước 4: Xóa chi tiết liên quan  
 deleteRelatedChiTiet(chiTietList);  
  
 // Bước 5: Xóa phiếu chính  
 deletePhieuRecord(phieuId);  
  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Delete Phieu Success", ""));  
 }  
  
 // Bước 1: Kiểm tra sự tồn tại của phiếu (phải được triển khai bởi lớp con)  
 protected abstract ResponseEntity<ResponseObject> checkPhieuExistence(Long phieuId);  
  
 // Bước 2: Tìm danh sách chi tiết liên quan (phải được triển khai bởi lớp con)  
 protected abstract Object findRelatedChiTiet(Long phieuId);  
  
 // Bước 3: Cập nhật số lượng kho (phải được triển khai bởi lớp con)  
 protected abstract ResponseEntity<ResponseObject> updateInventory(Object chiTietList);  
  
 // Bước 4: Xóa chi tiết liên quan (phải được triển khai bởi lớp con)  
 protected abstract void deleteRelatedChiTiet(Object chiTietList);  
  
 // Bước 5: Xóa phiếu chính (phải được triển khai bởi lớp con)  
 protected abstract void deletePhieuRecord(Long phieuId);

public class PhieuNhapDeletion extends AbstractPhieuDeletion {  
 @Autowired  
 private PhieuNhapRepository phieuNhapRepository;  
 @Autowired  
 private ChiTietPNRepository chiTietPNRepository;  
 @Autowired  
 private QuanLySPService quanLySPService;  
  
 @Override  
 protected ResponseEntity<ResponseObject> checkPhieuExistence(Long phieuId) {  
 Optional<PhieuNhap> foundPN = phieuNhapRepository.findById(phieuId);  
 if (!foundPN.isPresent()) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "Can't find PhieuNhap with id = " + phieuId, ""));  
 }  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(null);  
 }  
  
 @Override  
 protected Object findRelatedChiTiet(Long phieuId) {  
 Optional<PhieuNhap> foundPN = phieuNhapRepository.findById(phieuId);  
 if (!foundPN.isPresent()) {  
 return null;  
 }  
 return chiTietPNRepository.findAllByPhieuNhap(foundPN);  
 }  
  
 @Override  
 protected ResponseEntity<ResponseObject> updateInventory(Object chiTietList) {  
 List<ChiTietPhieuNhap> boxCTPN = (List<ChiTietPhieuNhap>) chiTietList;  
 return quanLySPService.updateEffectByDeleteChiTietPN(boxCTPN);  
 }  
  
 @Override  
 protected void deleteRelatedChiTiet(Object chiTietList) {  
 List<ChiTietPhieuNhap> boxCTPN = (List<ChiTietPhieuNhap>) chiTietList;  
 chiTietPNRepository.deleteAll(boxCTPN);  
 }  
  
 @Override  
 protected void deletePhieuRecord(Long phieuId) {  
 phieuNhapRepository.deleteById(phieuId);  
 }  
}

Tại PhieuNhapService:

public ResponseEntity<ResponseObject> deletePhieuNhap(Long phieuNhapId) {  
 return deletionProcess.deletePhieu(phieuNhapId);  
}

## Iterator Pattern

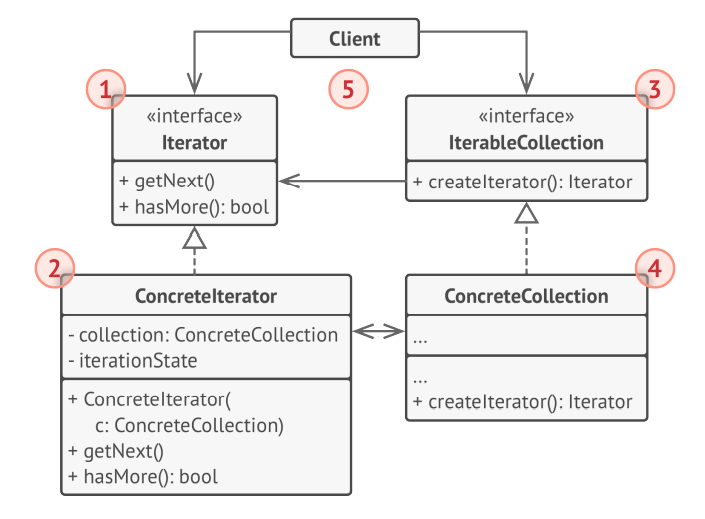
### Cơ sở lý thuyết

#### Khái niệm

Iterator hay còn gọi là Cursor là một mẫu thiết kế thuộc nhóm Hành vi (Behavioral Pattern). Mục tiêu của Iterator là cung cấp một cách thức chuẩn để truy cập tuần tự các phần tử trong một tập hợp (collection) mà không cần tiết lộ cấu trúc bên trong của nó. Iterator giúp tách biệt trách nhiệm giữa lưu trữ dữ liệu và duyệt qua dữ liệu, tuân thủ Nguyên tắc Single Responsibility Principle (SRP) trong SOLID.

Ngoài ra, Iterator còn tăng tính đóng gói, giúp client có thể truy cập các phần tử của một tập hợp theo nhiều cách khác nhau mà không cần sửa đổi cấu trúc của chính tập hợp đó. Điều này cũng giúp dễ dàng áp dụng nguyên tắc Open/Closed – cho phép mở rộng hành vi mà không thay đổi mã nguồn có sẵn.

#### Nội dung



Hình . Sơ đồ lớp Iterator Pattern

Iterator Pattern bao gồm các thành phần chính:

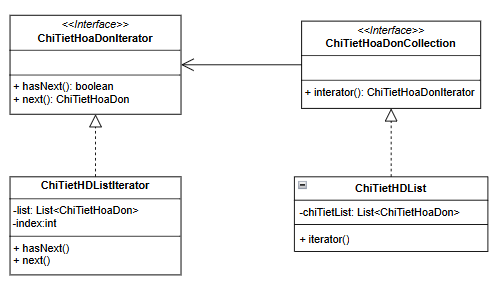
* **Iterator** : là một interface hay abstract class, khai báo các hoạt động cần thiết để tra so sánh một tập hợp: tìm nạp phần tử tiếp theo, truy xuất vị trí hiện tại, bắt đầu lại lặp lại, v.v.
* **Concrete** **Iterator** : implement các phương thức của Iterator, giữ index khi duyệt qua các phần tử. Cho phép một số trình vòng lặp đi qua cùng một bộ sưu tập độc lập với nhau.
* **Collection** **Interface**: khai báo một hoặc nhiều phương thức để nhận được các Iterator tương thích với collection. Lưu ý rằng kiểu trả về của các phương thức phải được khai báo dưới dạng Iterator Interface để các collection cụ thể có thể trả về các các loại Iterator.
* **Concrete** **Collections**: trả về các phiên bản mới của một lớp Concrete Iterator cụ thể mỗi khi client yêu cầu.
* **Client** : đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

### Lý do áp dụng

Thông thường, khi xóa hóa đơn, ta cần duyệt qua danh sách các ChiTietHoaDon để thực hiện các hành động như cập nhật lại số lượng sách trong kho, xóa từng chi tiết, đồng thời cập nhật tổng tiền của hóa đơn.

Để tách biệt logic duyệt danh sách và thao tác xóa, tăng tính mở rộng, em đã quyết định áp dụng Iterator Pattern để duyệt qua danh sách ChiTietHoaDon một cách linh hoạt và có thể tái sử dụng ở nhiều nơi khác.

### Sơ đồ lớp



### Code áp dụng

public class ChiTietHDList implements ChiTietHoaDonCollection {  
 private final List<ChiTietHoaDon> chiTietList;  
  
 public ChiTietHDList(List<ChiTietHoaDon> chiTietList) {  
 this.chiTietList = chiTietList;  
 }  
  
 @Override  
 public ChiTietHoaDonIterator iterator() {  
 return new ChiTietHDListIterator(chiTietList);  
 }  
  
}

public class ChiTietHDListIterator implements ChiTietHoaDonIterator {  
  
 private final List<ChiTietHoaDon> list;  
 private int index = 0;  
  
 public ChiTietHDListIterator(List<ChiTietHoaDon> list) {  
 this.list = list;  
 }  
  
 @Override  
 public boolean hasNext() {  
 return index < list.size();  
 }  
  
 @Override  
 public ChiTietHoaDon next() {  
 return list.get(index++);  
 }  
}

package vn.tdtu.finalterm.service.iterator;  
  
public interface ChiTietHoaDonCollection {  
 ChiTietHoaDonIterator iterator();  
}

public interface ChiTietHoaDonIterator extends Iterator<ChiTietHoaDon> {  
 boolean hasNext();  
 ChiTietHoaDon next();  
}

Tại ChiTietHDService:

public ResponseEntity<ResponseObject> deleteHoaDon(Long hoaDonId) {  
 Optional<HoaDon> foundHD = hoaDonRepository.findById(hoaDonId);  
  
 if(!foundHD.isPresent()) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "Can't find HoaDon with id = " + hoaDonId, "")  
 );  
 }  
  
 // Find All ChiTietHoaDon relevant to Delete  
 List<ChiTietHoaDon> boxCTHD = chiTietHDRepository.findAllByHoaDon(foundHD);  
  
 // Get some info for parameter  
 Long[] boxSPId = new Long[boxCTHD.size()];  
 ChiTietHoaDon[] boxCTHDterm = new ChiTietHoaDon[boxCTHD.size()];  
 int index = 0;  
 ChiTietHoaDonCollection collection = new ChiTietHDList(boxCTHD);  
 ChiTietHoaDonIterator iterator = collection.iterator();  
 while (iterator.hasNext()) {  
 ChiTietHoaDon item = iterator.next();  
 boxSPId[index] = item.getSanPham().getId();  
 boxCTHDterm[index] = new ChiTietHoaDon(null, 0, item.getSoLuong() \* -1, 0, null, null, null);  
 index++;  
 }  
  
 // Change quantity for QuanLySP  
 ResponseEntity<ResponseObject> checkQuantity = quanLySPService.deleteQuantityQuanLySP(boxSPId, foundHD.get().getChiNhanh().getId(), boxCTHDterm);  
 if(checkQuantity != null) {  
 return checkQuantity;  
 }  
  
 // Delete All ChiTietHoaDon  
 chiTietHDRepository.deleteAll(boxCTHD);  
  
 // Delete  
 hoaDonRepository.deleteById(hoaDonId);  
  
 // Thông báo sự kiện "HOADON\_DELETED"  
 notifyListeners("HOADON\_DELETED", foundHD.get());  
  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Delete HoaDon Success", "")  
 );  
}

## Observer Pattern

### Cơ sở lý thuyết

#### Khái niệm

Observer Pattern là mẫu thiết kế dùng để định nghĩa một cơ chế đăng kí (subcribe) nhằm thông báo (notify) cho nhiều đối tượng về các sự kiện xảy ra với đối tượng mà chúng đang quan sát (observer).

#### Nội dung



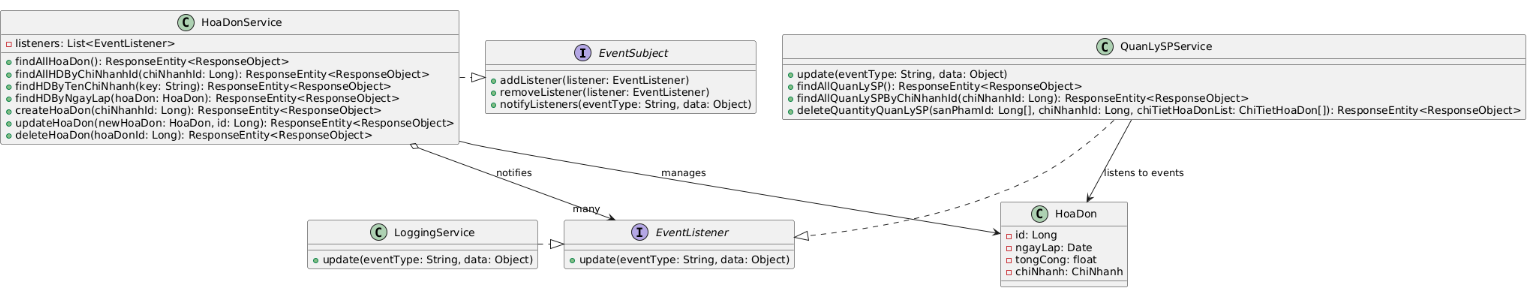
* Command: Interface hoặc Abtract class chứa một phương thức excecute() để thực thi request.
* ConcreteCommand: là các implementation của Command. Mỗi Concrete Command thực hiện một nhiệm vụ cụ thể khi được gọi phương thức execute().
* Invoker: lớp chứa đối tượng Command và gọi phương thức execute() của nó khi cần thiết, không quan tâm đến chi tiết của Command , chỉ quan tâm đến việc gọi execute() để thực thi request.
* Receiver: thành phần xử lý logic khi một request gửi đến. Trong phương thức execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver để thực hiện công việc cụ thể.
* Client: tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó. Client chịu trách nhiệm khởi tạo và cấu hình các Concrete Command trước khi chúng được gửi đến Invoker để thực thi.

### Lý do áp dụng

Observer Pattern được sử dụng trong trường hợp này để đồng bộ hóa các thay đổi giữa các thành phần liên quan (hóa đơn và quản lý sản phẩm) một cách linh hoạt và tách rời.

Các thao tác trong HoaDonService tạo ra các sự kiện cần thông báo cho các thành phần như QuanLySPService (cập nhật kho), hoặc các service khác (ghi log, báo cáo). Observer Pattern giúp tách biệt trách nhiệm và tăng tính linh hoạt.

### Sơ đồ lớp



### Code áp dụng

public interface EventListener {  
 void update(String eventType, Object data);  
}

public interface EventSubject {  
 void addListener(EventListener listener);  
 void removeListener(EventListener listener);  
 void notifyListeners(String eventType, Object data);  
}

Tại HoaDonService:

public ResponseEntity<ResponseObject> updateHoaDon(HoaDon newHoaDon, Long id) {  
 Optional<HoaDon> foundHD = hoaDonRepository.findById(id);  
  
 if(!foundHD.isPresent()) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "Can't find HoaDon with id = " + id, "")  
 );  
 }  
  
 HoaDon updatedHD = foundHD  
 .map(hoaDon -> {  
 hoaDon.setNgayLap(newHoaDon.getNgayLap());  
  
 return hoaDonRepository.save(hoaDon);  
 }).orElseGet(() -> {  
 return null;  
 });  
  
 // Thông báo sự kiện "HOADON\_UPDATED"  
 notifyListeners("HOADON\_UPDATED", updatedHD);  
  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Update HoaDon Success", updatedHD)  
 );  
}  
  
public ResponseEntity<ResponseObject> deleteHoaDon(Long hoaDonId) {  
 Optional<HoaDon> foundHD = hoaDonRepository.findById(hoaDonId);  
  
 if(!foundHD.isPresent()) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "Can't find HoaDon with id = " + hoaDonId, "")  
 );  
 }  
  
 // Find All ChiTietHoaDon relevant to Delete  
 List<ChiTietHoaDon> boxCTHD = chiTietHDRepository.findAllByHoaDon(foundHD);  
  
 // Get some info for parameter  
 Long[] boxSPId = new Long[boxCTHD.size()];  
 ChiTietHoaDon[] boxCTHDterm = new ChiTietHoaDon[boxCTHD.size()];  
 int index = 0;  
 ChiTietHoaDonCollection collection = new ChiTietHDList(boxCTHD);  
 ChiTietHoaDonIterator iterator = collection.iterator();  
 while (iterator.hasNext()) {  
 ChiTietHoaDon item = iterator.next();  
 boxSPId[index] = item.getSanPham().getId();  
 boxCTHDterm[index] = new ChiTietHoaDon(null, 0, item.getSoLuong() \* -1, 0, null, null, null);  
 index++;  
 }  
  
 // Change quantity for QuanLySP  
 ResponseEntity<ResponseObject> checkQuantity = quanLySPService.deleteQuantityQuanLySP(boxSPId, foundHD.get().getChiNhanh().getId(), boxCTHDterm);  
 if(checkQuantity != null) {  
 return checkQuantity;  
 }  
  
 // Delete All ChiTietHoaDon  
 chiTietHDRepository.deleteAll(boxCTHD);  
  
 // Delete  
 hoaDonRepository.deleteById(hoaDonId);  
  
 // Thông báo sự kiện "HOADON\_DELETED"  
 notifyListeners("HOADON\_DELETED", foundHD.get());  
  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Delete HoaDon Success", "")  
 );  
}  
  
public ResponseEntity<ResponseObject> createHoaDon(Long chiNhanhId) {  
 Optional<ChiNhanh> chiNhanh = chiNhanhRepository.findById(chiNhanhId);  
 if(!chiNhanh.isPresent()) {  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(  
 new ResponseObject("failed", "Can't find ChiNhanh with id = " + chiNhanhId, "")  
 );  
 }  
  
 HoaDon hoaDon = (HoaDon) phieuFactoryProvider.getFactory("hoadon").createPhieu(chiNhanh.get());  
 HoaDon savedHoaDon = hoaDonRepository.save(hoaDon);  
  
 // Thông báo sự kiện "HOADON\_CREATED"  
 notifyListeners("HOADON\_CREATED", savedHoaDon);  
  
 return ResponseEntity.*status*(HttpStatus.*OK*).body(  
 new ResponseObject("ok", "Create HoaDon Success", savedHoaDon)  
 );  
}

Tại QuanLySPService:

public class QuanLySPService implements EventListener {  
 @Autowired  
 QuanLySPRepository quanLySPRepository;  
 @Autowired  
 ChiNhanhRepository chiNhanhRepository;  
 @Autowired  
 SanPhamRepository sanPhamRepository;  
 @Autowired  
 HoaDonService hoaDonService;  
 public QuanLySPService(HoaDonService hoaDonService) {  
 this.hoaDonService = hoaDonService;  
 hoaDonService.addListener(this);  
 }  
 @Override  
 public void update (String eventType, Object data){  
 if (eventType.equals("HOADON\_CREATED") && data instanceof HoaDon) {  
 HoaDon hoaDon = (HoaDon) data;  
 System.*out*.println("Hóa đơn mới được tạo: " + hoaDon.getId() + ". Cập nhật kho...");  
  
 } else if (eventType.equals("HOADON\_DELETED")) {  
 System.*out*.println("Hóa đơn bị xóa: " + ((HoaDon) data).getId() + ". Khôi phục kho...");  
 }  
 }

## State Pattern

### Cơ sở lý thuyết

#### Khái niệm

State Pattern là một mẫu thiết kế thuộc nhóm mẫu thiết kế hành vi được áp dụng nhằm thay đổi hành vi của một đối tượng dựa trên trạng thái bên trong khi trạng thái của đối tượng ấy thay đổi. Nhờ việc áp dụng mẫu thiết kế này sẽ giúp việc giao tiếp giữa các đối tượng sẽ trở nên linh hoạt và ít gặp vấn đề lỗi phát sinh hơn.

#### Nội dung

A diagram of a computer

AI-generated content may be incorrect.

* Interface State là interface được định nghĩa đầu tiên nhằm xác định các phương thức đặc tính cơ bản của đối tượng giúp đảm bảo tính nhất quán.
* ConcreateState là các lớp được cài đặt cụ thể các phương thức đã được xác định ở State. Tại đây mỗi lớp ConcreteState đại diện cho một trạng thái cụ thể sẽ thực hiện logic và hành vi riêng tùy thuộc vào Context đó.
* Context là lớp được tạo ra để Client truy vấn đến. Client sẽ không truy vấn trực tiếp đến trạng thái của đối tượng mà sẽ gọi thông qua lớp Context trên. Lớp Context sẽ chứa thông tin các ConcreteState của đối tượng.

Lưu ý: Hành vi của đối tượng sẽ thay đổi khi trạng thái thay đổi, điều này cần được thực hiện rõ ràng và nhất quán thông qua các lớp ConcreteState.

### Lý do áp dụng

Nhằm mục đích mở rộng các trạng thái của TaiKhoan và hạn chế việc phức tạp hóa khi định nghĩa nhiều trạng thái tài khoản, việc áp dụng State Pattern sẽ làm tối ưu hóa và tinh gọn code hơn, giúp dễ đọc cũng như cập nhật phát triển các trạng thái tài khoản sau này, ví dụ như trạng thái khóa tài khoản, tài khoản bị xóa,...

### Sơ đồ lớp

A diagram of a computer

AI-generated content may be incorrect.

### Code áp dụng

A screen shot of a computer

AI-generated content may be incorrect.

Khởi tạo interface AccountState định nghĩa phương thức cho các ConcreteState. Ở ví dụ trên em viết hai phương thức chính là handleState() nhằm xử lý các trạng thái và phương thức getStateName() để lấy tên của trạng thái.

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

Hai lớp UnverifiedState và VerifiedState chính là hai lớp ConcreteState implement interface AccountState nhằm định nghĩa hai hành vi chưa xác thực và xác thực và thực hiện xử lý các hàm đã định nghĩa ở interface.

A computer screen with colorful text

AI-generated content may be incorrect.

Ở model TaiKhoan bổ sung thêm hàm setState() để các lớp concreteState truy vấn đến giúp quản lý việc chuyển đổi trạng thái. Đây chính là lớp Concrete để truy vấn đến.

A screen shot of a computer program

AI-generated content may be incorrect.

Tại lớp TaiKhoanService ở hàm verifyRegistration() thực hiện chuyển đổi trạng thái xác thực hoặc chưa xác thực của tài khoản sau khi đăng kí tài khoản và chi nhánh.

A screen shot of a computer

AI-generated content may be incorrect.

Tương tự tại lớp ChiNhanhService tại phương thức insertChiNhanh thực khởi tạo tài khoản ở trạng thái chưa xác thực, gửi email xác thực và sau khi xác thực sẽ chuyển đổi trạng thái sang xác thực cho tài khoản liên kết với chi nhánh vừa được tạo.

## Command Pattern

### Cơ sở lý thuyết

#### Khái niệm

Command Pattern là mẫu thiết kế thuộc mẫu thiết kế hành vi nhằm cho phép chuyển đổi một request thành một đối tượng độc lập chứa thông tin của request đó. Nhằm tách biệt dễ dàng cho việc gọi lệnh thực thi cũng như bảo trì cập nhật code.

#### Nội dung

A diagram of a server

AI-generated content may be incorrect.

* Command: Đây là interface định nghĩa phương thức ban đầu để các lớp implement cài đặt.
* ConcreteCommand: Đây là các lớp implement từ interface Command để cài đặt hàm được định nghĩa trên interface. Nhằm liên kết giữa Receiver và 1 hành động.
* Client: Đây là lớp nhận request từ user, từ đó thiết lập receiver bằng việc đóng gói request thành các ConcreateCommand thích hợp.
* Invoker: Lớp này sẽ tiếp nhận ConcreteCommand từ Client và gọi hàm thực thi.
* Receiver xử lý logic cho các request.

### Lý do áp dụng

Việc áp dụng Command Pattern giúp đóng gói các thao tác CRUD giúp code trở nên tinh gọn, logic. Đồng thời tăng khả năng mở rộng khi muốn thêm các thao tác khác chỉ cần chỉnh sửa lại code ở pattern mà không phải viết phức tạp ở nhiều file. Đồng thời tăng khả năng tái sử dụng code và dễ dàng bảo trì cũng như cập nhật.

### Sơ đồ lớp

A diagram of a computer

AI-generated content may be incorrect.

### Code áp dụng

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

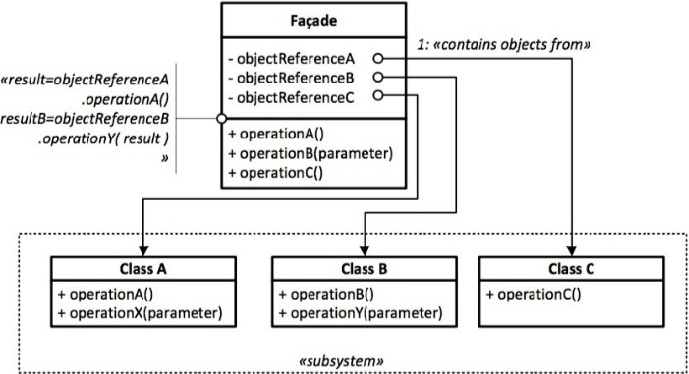
## Facade Pattern

### Cơ sở lý thuyết

#### Khái niệm

Facade Pattern là mẫu thiết kế thuộc mẫu thiết kế nhóm cấu trúc. Đây là loại mẫu thiết kế nhằm cung cấp cho ta một giao diện chung, từ đó cho phép truy cập trực tiếp đến giao diện chung này để giao tiếp với các giao diện có trong hệ thống con.

#### Nội dung



* Lớp Facade là lớp giao diện cơ bản thống nhất truy cập vào tập hợp các lớp trong hệ thống phức tạp.
* Các thuộc tính objectReference tham chiếu đến các đối tượng tương ứng của lớp A, B, C và các phương thước tại lớp Facade cũng tham chiếu đến các phương thức của các lớp A, B, C.
* Từ đó khi Client gọi phương thức trong Facade, lớp này sẽ khởi tạo và sử dụng các đối tượng con và gọi các phương thức tương ứng tuần tự

### Lý do áp dụng

Trong quá trình thực hiện lập hóa đơn, việc tạo lập hóa đơn vô cùng phức tạp khi bao hàm nhiều bước từ hóa đơn đến chi tiết hóa đơn, việc áp dụng Facade Pattern vào quy trình này sẽ gói gọn lại trong một lớp Facade từ đó giảm thiếu độ phức tạp của chức năng, dễ dàng kiểm tra sửa lỗi nếu có cũng như giảm thiếu sự phụ thuộc giữa lớp Controller và lớp Service.

### Sơ đồ lớp

A computer screen shot of a computer program

AI-generated content may be incorrect.

### Code áp dụng

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

## Factory Pattern

### Cơ sở lý thuyết

#### Khái niệm

Factory Pattern là mẫu thiết kế thuộc mẫu thiết kế khởi tạo. Mẫu thiết kế này cho phép người dùng khởi tạo đối tượng của lớp mà không cần phải chỉ rõ lớp cụ thể nào sẽ được khởi tạo.

#### Nội dung

A diagram of a product

AI-generated content may be incorrect.

* Product: Đây là lớp khuôn mẫu interface của đối tượng mà Factory sẽ tạo ra.
* ConcreteProduct là các lớp được cài đặt từ lớp Product.
* Creator là lớp khai báo factory method, trả về đối tượng thuộc kiểu product.
* ConcreteCreator: Đây là lớp ghi đè factory để trả về instance của ConcreteCroduct

### Lý do áp dụng

Việc áp dụng Factory Pattern vào dự án nhằm tách biệt logic tạo đối tượng phức tập như Phiếu và Hóa đơn. Giúp đơn giản hóa từ đó dễ dàng cho việc mở rộng ví dụ khi cần thêm một loại phiếu mới chỉ cần tạo factory mới cho loại phiếu đó mà không cần phải chỉnh sửa quá phức tạp. Từ đó giảm thiểu sự phụ thuộc code tăng tính linh hoạt cho việc cập nhật, sửa chữa bảo trì code.

### Sơ đồ lớp

A screenshot of a computer

AI-generated content may be incorrect.

### Code áp dụng

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

.

# KẾT LUẬN

Trong quá trình thực hiện dự án áp dựng mẫu thiết kế vào website quản lý nhà sách, việc áp dụng các mẫu thiết kế đã giúp tăng cường tính linh hoạt, tái sử dụng, tăng khả năng mở rộng cũng như dễ dàng bảo trì cập nhật.

# TÀI LIỆU THAM KHẢO

Tiếng Việt

Tiếng Anh