**BỘ THÔNG TIN VÀ TRUYỀN THÔNG**

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG CƠ SỞ TẠI THÀNH PHỐ HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN II**

A logo with a star

AI-generated content may be incorrect.

BÁO CÁO BÀI TẬP

**XÂY DỰNG CÁC MẪU THIẾT KẾ FACTORY, SINGLETON CHO HỆ THỐNG BÁN LAPTOP ONLINE**

|  |  |  |
| --- | --- | --- |
| **Giảng viên hướng dẫn** | **Th.S Nguyễn Văn Hữu Hoàng** |  |
| **Sinh viên thực hiện** | **Nguyễn Đình Luật** | **N21DCCN145** |
|  | **Lê Tuấn Nhật** | **N21DCCN160** |
|  | **Trần Bá Thọ** | **N21DCCN181** |
| **Lớp** | **D21CQCNPM02-N** |  |
|  |  |  |
|  |  |  |
|  |  |  |
| ***Thành phố Hồ Chí Minh, ngày 19 tháng 4 năm 2025*** | | |

# LỜI GIỚI THIỆU

Đề tài **“Xây dựng các mẫu thiết kế Factory, Singleton cho hệ thống bán laptop online”** hướng đến việc ứng dụng các *mẫu thiết kế (design pattern)* kinh điển vào một hệ thống phần mềm thực tế. Cụ thể, hệ thống bán laptop trực tuyến sẽ được thiết kế hướng đối tượng, trong đó mẫu **Factory** được sử dụng để tạo ra các đối tượng Laptop một cách linh hoạt, còn mẫu **Singleton** được sử dụng để quản lý **lịch sử đơn hàng** tập trung.

Mục tiêu của đề tài bao gồm:

* **Hiểu và áp dụng được mẫu thiết kế Factory và Singleton**: nhóm phát triển cần nắm vững nguyên lý của hai mẫu thiết kế này và tích hợp chúng vào hệ thống.
* **Xây dựng hệ thống bán laptop online cơ bản**: hệ thống cho phép người dùng xem danh mục sản phẩm, thêm sản phẩm vào giỏ hàng, đặt hàng và xem lịch sử đơn hàng.
* **Cải thiện cấu trúc và khả năng mở rộng của hệ thống**: việc áp dụng design pattern giúp mã nguồn dễ bảo trì, dễ mở rộng và tái sử dụng. Đặc biệt, khi bổ sung tính năng hoặc loại sản phẩm mới, kiến trúc có sẵn sẽ giảm thiểu thay đổi trên diện rộng.

**MỤC LỤC**

[**LỜI GIỚI THIỆU 2**](#_Toc195923569)

[**CHƯƠNG 1: PHÂN TÍCH YÊU CẦU VÀ CHỨC NĂNG CƠ BẢN CỦA HỆ THỐNG 4**](#_Toc195923570)

[**CHƯƠNG 2: PHÂN TÍCH HỆ THỐNG 6**](#_Toc195923571)

[**2.1. Cấu trúc hệ thống 6**](#_Toc195923572)

[**2.2. Luồng xử lý 7**](#_Toc195923573)

[**2.2.1. Hiển thị danh mục sản phẩm: 7**](#_Toc195923574)

[**2.2.2. Thêm sản phẩm vào giỏ hàng: 7**](#_Toc195923575)

[**2.2.3. Xem giỏ hàng: 7**](#_Toc195923576)

[**2.2.4. Đặt hàng: 7**](#_Toc195923577)

[**2.2.5. Xem lịch sử đơn hàng: 7**](#_Toc195923578)

[**2.3. Áp dụng các mẫu thiết kế 7**](#_Toc195923579)

[**2.3.1. Mẫu Factory trong việc khởi tạo Laptop 7**](#_Toc195923580)

[**2.3.2. Mẫu Singleton trong quản lý lịch sử đơn hàng 8**](#_Toc195923581)

[**2.4. Các đoạn code minh họa các mẫu thiết kế 9**](#_Toc195923582)

[**2.4.1. Code triển khai Factory – ProductFactory.java 9**](#_Toc195923583)

[**2.4.2. Code implement của Product, ví dụ GamingLaptop.Java 9**](#_Toc195923584)

[**2.4.3. Code lớp Factory để tạo ra các Product 9**](#_Toc195923585)

[**2.4.4. Code triển khai Singleton – OrderHistory.java 10**](#_Toc195923586)

[**2.4.5. Ví dụ về cách sử dụng Singleton trong đoạn code đặt hàng 10**](#_Toc195923587)

[**2.5. Ưu điểm và nhược điểm của việc áp dụng các mẫu thiết kế này 10**](#_Toc195923588)

[**2.5.1. Mẫu Factory 11**](#_Toc195923589)

[**2.5.2. Mẫu Singleton 11**](#_Toc195923590)

# PHÂN TÍCH YÊU CẦU VÀ CHỨC NĂNG CƠ BẢN CỦA HỆ THỐNG

Hệ thống bán laptop online là một ứng dụng mô phỏng hoạt động của một cửa hàng trực tuyến, được thiết kế để đáp ứng các yêu cầu cơ bản của người dùng trong việc mua sắm trực tuyến.

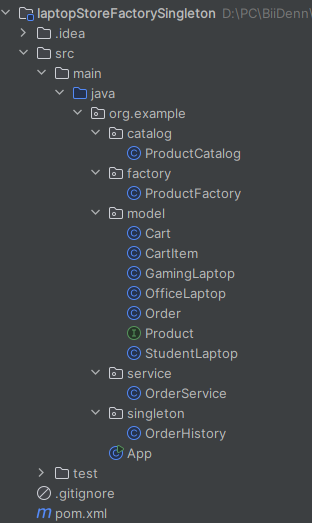
* **Yêu cầu hệ thống:**
* **Hiển thị danh mục sản phẩm:** Người dùng có thể xem danh sách các loại laptop có sẵn, bao gồm Gaming Laptop (dành cho chơi game), Office Laptop (dành cho công việc văn phòng) và Student Laptop (dành cho học sinh, sinh viên). Mỗi sản phẩm sẽ đi kèm thông tin cơ bản như tên, giá và mô tả ngắn gọn.
* **Quản lý giỏ hàng:** Người dùng có thể chọn sản phẩm từ danh mục, chỉ định số lượng mong muốn và thêm vào giỏ hàng. Hệ thống cần hiển thị nội dung giỏ hàng, bao gồm danh sách sản phẩm đã chọn, số lượng và tổng giá trị đơn hàng.
* **Đặt hàng:** Sau khi hoàn tất việc chọn sản phẩm, người dùng có thể đặt hàng dựa trên nội dung giỏ hàng. Hệ thống sẽ tạo một đơn hàng mới và lưu trữ thông tin đơn hàng vào lịch sử.
* **Xem lịch sử đơn hàng:** Người dùng có thể truy cập lịch sử để xem lại tất cả các đơn hàng đã đặt trước đó, bao gồm chi tiết về sản phẩm, số lượng và thời gian đặt hàng.
* **Vai trò của các mẫu thiết kế:**
* **Mẫu Factory (ProductFactory):** Được sử dụng để tạo các đối tượng sản phẩm (laptop) mà không cần chỉ định trực tiếp lớp cụ thể của chúng. Ví dụ, khi người dùng chọn "Gaming Laptop", ProductFactory sẽ tạo một instance của lớp GamingLaptop. Điều này giúp hệ thống dễ dàng bổ sung các loại laptop mới (như "Business Laptop") trong tương lai chỉ bằng cách thêm lớp mới và cập nhật factory, mà không cần thay đổi các phần khác của mã nguồn.
* **Mẫu Singleton (OrderHistory):** Đảm bảo rằng chỉ có một instance duy nhất của lớp OrderHistory tồn tại trong toàn bộ hệ thống. Điều này rất quan trọng để quản lý lịch sử đơn hàng một cách tập trung, tránh tình trạng dữ liệu bị phân mảnh hoặc không đồng bộ giữa các phần khác nhau của ứng dụng.
* **Tương tác giữa các thành phần:**
* **Xem danh mục:** Người dùng truy cập ProductCatalog để xem danh sách sản phẩm. ProductCatalog đóng vai trò như một kho lưu trữ danh mục, cung cấp thông tin để hiển thị cho người dùng.
* **Thêm sản phẩm vào giỏ hàng:** Khi người dùng chọn một sản phẩm, hệ thống gọi ProductFactory.createProduct(type) để tạo đối tượng sản phẩm tương ứng. Sau đó, sản phẩm được thêm vào Cart dưới dạng CartItem, bao gồm thông tin về sản phẩm và số lượng.
* **Đặt hàng:** Khi người dùng xác nhận đặt hàng, OrderService sẽ lấy thông tin từ Cart, tạo một Order mới và gửi nó đến OrderHistory để lưu trữ.
* **Xem lịch sử:** Người dùng có thể yêu cầu hiển thị lịch sử đơn hàng bất kỳ lúc nào, và OrderHistory sẽ cung cấp danh sách các đơn hàng đã lưu.

Hệ thống được thiết kế để đảm bảo tính trực quan và dễ sử dụng, đồng thời tận dụng tối đa các mẫu thiết kế để tăng tính linh hoạt và khả năng bảo trì.

# PHÂN TÍCH HỆ THỐNG

## Cấu trúc hệ thống

Hệ thống được xây dựng theo kiến trúc phân tầng, với các thành phần được chia thành các package riêng biệt để đảm bảo tính module hóa:



* **model:** Bao gồm các lớp như Product (lớp cơ sở), GamingLaptop, OfficeLaptop, StudentLaptop (các lớp con), Cart, CartItem và Order. Đây là nơi định nghĩa dữ liệu và logic nghiệp vụ cơ bản.
* **factory:** Chứa lớp ProductFactory, chịu trách nhiệm tạo các đối tượng sản phẩm dựa trên loại sản phẩm được yêu cầu.
* **service:** Chứa OrderService, xử lý logic nghiệp vụ liên quan đến việc tạo và quản lý đơn hàng.
* **singleton:** Chứa OrderHistory, sử dụng mẫu Singleton để quản lý lịch sử đơn hàng.
* **catalog:** Chứa ProductCatalog, chịu trách nhiệm lưu trữ và hiển thị danh mục sản phẩm.

## Luồng xử lý

Quy trình hoạt động của hệ thống bao gồm các bước sau:

### Hiển thị danh mục sản phẩm:

* + ProductCatalog chứa danh sách sản phẩm dưới dạng List<Product> và cung cấp phương thức để hiển thị danh sách này cho người dùng.

### Thêm sản phẩm vào giỏ hàng:

* + Người dùng chọn loại sản phẩm và số lượng. Hệ thống gọi ProductFactory.createProduct(type) để tạo đối tượng sản phẩm.
  + Đối tượng sản phẩm được thêm vào Cart thông qua phương thức addProduct(product, quantity), tạo một CartItem mới.

### Xem giỏ hàng:

* + Cart sử dụng phương thức printCart() để hiển thị danh sách CartItem cùng tổng giá trị đơn hàng.

### Đặt hàng:

* + OrderService lấy dữ liệu từ Cart, tạo một Order mới và gọi OrderHistory.addOrder(order) để lưu trữ đơn hàng.

### Xem lịch sử đơn hàng:

* + Người dùng truy cập OrderHistory.getInstance().printHistory() để xem danh sách các đơn hàng đã đặt.

## Áp dụng các mẫu thiết kế

### Mẫu Factory trong việc khởi tạo Laptop

**Factory Pattern** là mẫu thiết kế tạo ra đối tượng gián tiếp thông qua một **“factory”** chung, thay vì trực tiếp dùng từ khóa new tại nhiều nơi. Mẫu này giúp **đóng gói** logic khởi tạo và **ẩn** đi lớp cụ thể của đối tượng được tạo từ phía người dùng.

Trong hệ thống bán laptop, mẫu Factory được áp dụng thông qua lớp ProductFactory. Thay vì rải rác trong code tạo các đối tượng GamingLaptop, OfficeLaptop bằng new, ta sử dụng ProductFactory.createProduct(String type). Ví dụ, khi cần tạo một laptop gaming, ta gọi:



Bên trong ProductFactory, phương thức createProduct sẽ **quyết định** tạo ra instance thuộc lớp cụ thể nào dựa trên tham số *type*. Nhờ Factory Pattern, **phần còn lại của hệ thống không cần biết đến sự tồn tại của các lớp con cụ thể** (GamingLaptop, OfficeLaptop); nếu sau này thêm loại BusinessLaptop chẳng hạn, chỉ cần bổ sung lớp đó và điều chỉnh ProductFactory mà không ảnh hưởng code nơi khác.

Mẫu Factory giúp việc mở rộng các dòng sản phẩm laptop trở nên dễ dàng và tập trung. Ví dụ, nếu mở rộng từ bán laptop sang bán cả phụ kiện, ta có thể có một Factory khác hoặc mở rộng ProductFactory để tạo thêm các loại Product khác nhau. Tất cả logic khởi tạo tập trung một chỗ giúp kiểm soát tốt hơn (nếu cần thay đổi cách khởi tạo – ví dụ dùng một pool đối tượng, ta chỉ cần thay đổi trong factory).

### Mẫu Singleton trong quản lý lịch sử đơn hàng

**Singleton Pattern** đảm bảo một lớp chỉ có duy nhất **một instance** trên toàn ứng dụng và cung cấp một điểm truy cập toàn cục đến instance đó.

Trong hệ thống, việc quản lý **lịch sử đơn hàng** là nhiệm vụ đòi hỏi tập trung một chỗ – toàn bộ chương trình chỉ cần *một danh sách chung* lưu tất cả các đơn hàng của người dùng. Nếu mỗi lần xem lịch sử mà tạo mới một đối tượng OrderHistory khác nhau thì các đơn hàng cũ sẽ không được lưu lại. Do đó, ta áp dụng Singleton cho lớp OrderHistory.

Cụ thể, OrderHistory được thiết kế để không cho phép tạo mới nhiều đối tượng. Lần đầu tiên OrderHistory.getInstance() được gọi, nó sẽ trả về đối tượng OrderHistory duy nhất (được khởi tạo sẵn). Mọi nơi khác gọi getInstance() cũng nhận được đúng đối tượng đó, do đó thêm đơn hàng hay xem lịch sử đều thao tác trên cùng một danh sách orders.

Trong quá trình đặt hàng, sau khi OrderService tạo xong một Order, hệ thống sẽ gọi:



để lưu đơn hàng mới vào lịch sử. Khi người dùng yêu cầu xem lịch sử, hệ thống gọiOrderHistory.getInstance().printHistory() để hiển thị tất cả đơn hàng đã lưu.

Đảm bảo tính nhất quán của dữ liệu lịch sử đơn hàng trên toàn hệ thống. Mẫu Singleton cũng đơn giản hóa việc truy cập, không cần truyền đối tượng OrderHistory qua nhiều lớp; bất cứ đâu cần có thể gọi OrderHistory.getInstance(). Ngoài ra, nó tiết kiệm bộ nhớ khi tránh được việc tạo nhiều bản sao không cần thiết cho cùng một dữ liệu lịch sử.

## Các đoạn code minh họa các mẫu thiết kế

### Code triển khai Factory – ProductFactory.java

A screenshot of a computer program

AI-generated content may be incorrect.

Interface **Product** định nghĩa các phương thức chung cho mọi sản phẩm. Lớp cụ thể bên dưới sẽ triển khai interface này:

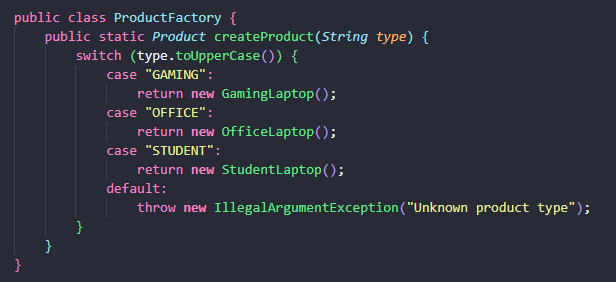
### Code implement của Product, ví dụ GamingLaptop.Java

A computer code with text

AI-generated content may be incorrect.

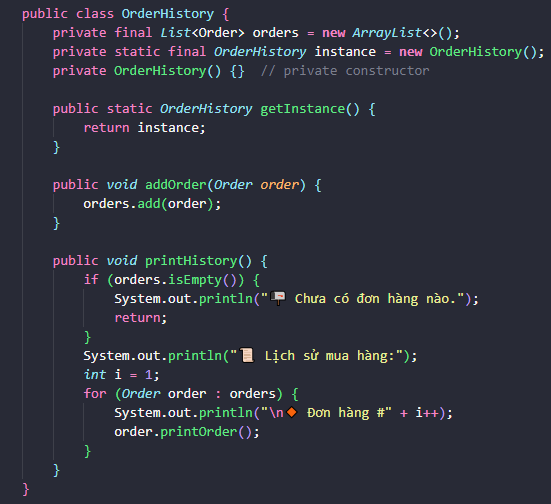
Tương tự OfficeLaptop, StudentLaptop triển khai Product với tên và giá khác.

### Code lớp Factory để tạo ra các Product

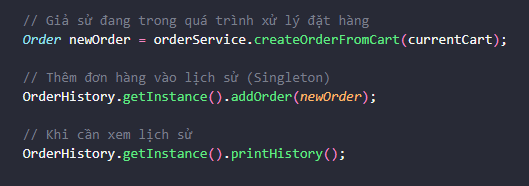


Với cách làm này, ở bất kỳ đâu trong mã nguồn muốn có đối tượng Laptop, ta chỉ việc gọi ProductFactory.createProduct("GAMING") hoặc loại tương ứng. Việc này thay thế cho các lệnh new GamingLaptop() rải rác, giúp mã nguồn **gọn gàng hơn** và tuân theo **nguyên tắc đóng mở**: thêm loại sản phẩm mới chỉ cần sửa nhẹ Factory (hoặc thậm chí có thể cải tiến Factory để tránh sửa khi thêm sản phẩm bằng cách dùng cấu hình bên ngoài).

### Code triển khai Singleton – OrderHistory.java



### Ví dụ về cách sử dụng Singleton trong đoạn code đặt hàng



## Ưu điểm và nhược điểm của việc áp dụng các mẫu thiết kế này

Việc áp dụng hai mẫu thiết kế **Factory** và **Singleton** đem lại nhiều ưu điểm cho hệ thống, nhưng cũng có những mặt hạn chế cần lưu ý. Dưới đây là phân tích ngắn gọn:

### Mẫu Factory

#### Ưu điểm:

* *Tách biệt việc tạo đối tượng khỏi logic sử dụng*: Các phần khác của hệ thống không cần biết chi tiết lớp cụ thể để tạo sản phẩm, chỉ cần yêu cầu Factory. Điều này giúp **giảm phụ thuộc** (loose coupling) giữa các thành phần.
* *Dễ mở rộng sản phẩm mới*: Thêm loại laptop mới ít ảnh hưởng đến mã nguồn hiện có. Chỉ cần bổ sung lớp mới và điều chỉnh nhẹ trong Factory (hoặc mở rộng Factory). Phần code sử dụng factory tự động hỗ trợ loại mới nếu được truyền tham số phù hợp.
* *Đóng gói tập trung*: Mọi logic khởi tạo được đặt tại ProductFactory, giúp bảo trì dễ dàng. Nếu khởi tạo sản phẩm trở nên phức tạp (ví dụ đọc từ file cấu hình hoặc tạo kèm các đối tượng liên quan), ta chỉ cần sửa trong Factory thay vì nhiều chỗ.

#### Nhược điểm:

* *Bổ sung sản phẩm mới vẫn cần sửa mã nguồn Factory*: Mặc dù ảnh hưởng hẹp, việc thêm một case mới trong createProduct nghĩa là vi phạm nguyên tắc **Open-Closed** ở mức độ nào đó (class Factory phải “mở” để chỉnh sửa khi có sản phẩm mới). Tuy nhiên, có thể khắc phục bằng cách dùng kỹ thuật khác như đăng ký (registry) sản phẩm động, nhưng sẽ phức tạp hơn.
* *Không phù hợp nếu chỉ có rất ít loại sản phẩm*: Nếu hệ thống chỉ có 1-2 loại và ít thay đổi, dùng Factory có thể là “thừa”, khiến code phức tạp không cần thiết. Trong bài toán này, chúng ta có nhiều loại laptop nên Factory là phù hợp.
* *Chi phí tạo thêm lớp Factory*: Làm tăng nhẹ số lượng lớp, nhưng đây không phải vấn đề lớn với lợi ích đạt được.

### Mẫu Singleton

#### Ưu điểm:

* *Đảm bảo duy nhất một thực thể quản lý trung tâm*: Với OrderHistory Singleton, toàn hệ thống luôn dùng chung một danh sách đơn hàng. Điều này **đảm bảo tính nhất quán** dữ liệu – mọi nơi đều thấy cùng thông tin lịch sử.
* *Truy cập thuận tiện*: Không cần truyền đối tượng OrderHistory qua các phương thức hay lưu dưới dạng biến toàn cục; bất kỳ phần nào của code khi cần có thể gọi OrderHistory.getInstance() để lấy đối tượng dùng ngay. Mẫu Singleton cung cấp một **điểm truy cập toàn cục** an toàn thay vì dùng biến global.
* *Tiết kiệm tài nguyên*: Thay vì tạo nhiều đối tượng quản lý lịch sử (mỗi cái có thể lưu một phần đơn hàng), ta chỉ dùng một đối tượng, giảm thiểu việc sử dụng bộ nhớ và xử lý khởi tạo.

#### Nhược điểm:

* *Tính toàn cục của Singleton có thể gây* ***phụ thuộc ngầm***: Do OrderHistory là dùng chung, nhiều phần của chương trình phụ thuộc vào nó như một trạng thái chung. Điều này nếu không quản lý tốt có thể dẫn tới **thiết kế kém linh hoạt** hoặc khó mở rộng cho các trường hợp phức tạp hơn (ví dụ: nếu sau này hệ thống hỗ trợ nhiều người dùng đăng nhập khác nhau, mỗi người có lịch sử đơn hàng riêng, thì Singleton kiểu toàn cục cho mọi người dùng sẽ không còn phù hợp).
* *Khó kiểm thử đơn vị*: Singleton giống như một biến toàn cục, nên khi viết unit test, rất khó thay thế hay mô phỏng (mock) nó bằng phiên bản khác. Nếu OrderHistory kết nối cơ sở dữ liệu chẳng hạn, các test cho phần sử dụng OrderHistory sẽ khó tách biệt.
* *Vấn đề đa luồng*: Trong môi trường đa luồng hoặc ứng dụng web nhiều request đồng thời, Singleton cần được cài đặt cẩn thận để tránh xung đột khi nhiều luồng cùng truy cập thay đổi dữ liệu. Với Java, cách tiếp cận ở trên (static instance) thường an toàn trong khởi tạo, nhưng thao tác trên dữ liệu orders bên trong vẫn cần cân nhắc đồng bộ (synchronization) nếu có nhiều luồng thêm đơn cùng lúc.
* *Khó mở rộng chức năng*: Vì Singleton gắn liền với ý tưởng một thể hiện duy nhất, nếu yêu cầu thay đổi (ví dụ muốn có 2 lịch sử riêng biệt, hoặc muốn reset lịch sử giữa chừng), việc này có thể khó thực hiện nếu thiết kế ban đầu không tính đến.