**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**

🙠🙟🕮🙝🙢

**BÁO CÁO DESIGN PATTERN**

SINGLETON, BUILDER & FACADE

**GIẢNG VIÊN HƯỚNG DẪN: ThS. Nguyễn Thị Thanh**

**MÃ MÔN HỌC: OOSD330879**

**NHÓM THỰC HIỆN: 15**

**SINH VIÊN THỰC HIỆN**

**Trần Chí Mỹ – 20110202**

**Phạm Ngọc Thắng – 20110728**

**Nguyễn Sinh Hùng – 20110647**

**Vũ Hoàng Anh – 20110205**

*Thủ Đức, tháng 5 năm 2023*

|  |  |
| --- | --- |
| Trường Đại học Sư phạm Kỹ thuật  TP. Hồ Chí Minh  Khoa Công Nghệ Thông Tin  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM  Độc lập – Tự do – Hạnh phúc  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**DANH SÁCH THÀNH VIÊN NHÓM 15  
HỌC KỲ II NĂM HỌC 2022-2023**

**1. Mã lớp môn học:** OOSD330879\_22\_2\_02

**2. Giảng viên hướng dẫn:** ThS. Nguyễn Thị Thanh

**3. Danh sách thành viên nhóm**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **HỌ VÀ TÊN** | **MSSV** | **ĐIỂM** |
| 1 | Trần Chí Mỹ | 20110202 |  |
| 2 | Phạm Ngọc Thắng | 20110728 |  |
| 3 | Vũ Hoàng Anh | 20110205 |  |
| 4 | Nguyễn Sinh Hùng | 20110647 |  |

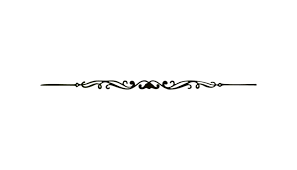
**Nhận xét của giảng viên:**

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

*Ngày……tháng……năm 2023*

*Giảng viên chấm điểm*

**MỤC LỤC**

[**NỘI DUNG** 1](#_Toc134826681)

[**I. Singleton Design Pattern(“Duy nhất”)** 1](#_Toc134826682)

[**1.1.** **Giới thiệu về Singleton Design Pattern** 1](#_Toc134826683)

[**1.2.** **Sơ đồ UML của Singleton Design Pattern** 2](#_Toc134826684)

[**1.3.** **Triển khai Singleton Design Pattern** 3](#_Toc134826685)

[**1.3.1.** **Lazy Initialized Singleton** 3](#_Toc134826686)

[**1.3.2.** **Eager Initialized Singleton** 4](#_Toc134826687)

[**1.3.3.** **Static block initialization** 4](#_Toc134826688)

[**1.3.4.** **Thread Safe Singleton** 5](#_Toc134826689)

[**1.3.5.** **Double Check Locking Singleton** 6](#_Toc134826690)

[**1.3.6.** **Reflection & Bill Pugh Singleton Implementation** 6](#_Toc134826691)

[**1.3.7.** **Enum Singleton** 7](#_Toc134826692)

[**1.4.** **Ưu điểm và nhược điểm của Singleton Design Pattern** 8](#_Toc134826693)

[**1.5.** **Demo** 9](#_Toc134826694)

[**1.5.1.** **Phạm Ngọc Thắng** 9](#_Toc134826695)

[**1.5.2.** **Nguyễn Sinh Hùng** 11](#_Toc134826696)

[**1.5.3. Vũ Hoàng Anh** 13](#_Toc134826697)

[**1.5.4. Trần Chí Mỹ** 16](#_Toc134826698)

[**II** **Builder Design Pattern(“Thợ Xây”)** 19](#_Toc134826699)

[**2.1.** **Giới thiệu về Builder Design Pattern** 19](#_Toc134826700)

[**2.2.** **Đặt vấn đề** 19](#_Toc134826701)

[**2.3.** **Đặc điểm của Builder Design Pattern** 20](#_Toc134826702)

[**2.4.** **Sơ đồ UML của Builder Design Pattern** 22](#_Toc134826703)

[**2.5.** **Ưu và nhược điểm của Builder Design Pattern** 23](#_Toc134826704)

[**2.6.** **Khi nào sử dụng Builder Design Pattern** 23](#_Toc134826705)

[**2.7.** **Demo** 24](#_Toc134826706)

[**2.7.1. Phạm Ngọc Thắng** 24](#_Toc134826707)

[**2.7.2. Nguyễn Sinh Hùng** 29](#_Toc134826708)

[**2.7.3. Vũ Hoàng Anh** 32](#_Toc134826709)

[**2.7.4. Trần Chí Mỹ** 35](#_Toc134826710)

[**III. Façade Design Pattern (“Mặt tiền”)** 38](#_Toc134826711)

[**3.1.** **Giới thiệu về Façade Design Pattern** 38](#_Toc134826712)

[**3.2.** **Đặt vấn đề** 38](#_Toc134826713)

[**3.3.** **Sơ đồ Diagram của Façade Design Pattern** 40](#_Toc134826714)

[**3.4.** **Triển khai Façade Design Pattern** 41](#_Toc134826715)

[**3.5.** **Ưu điểm và nhược điểm của Façade Design Pattern** 42](#_Toc134826716)

[**3.6.** **Demo** 43](#_Toc134826717)

[**TÀI LIỆU THAM KHẢO**](#_Toc134826718)

**DANH MỤC HÌNH ẢNH**

[Hình 1.1: Kiến trúc của Singleton Design Pattern 2](#_Toc134823324)

[Hình 1.2: Triển khai Lazy Initialized 3](#_Toc134823325)

[Hình 1.3: Triển khai Eager Initialized 4](#_Toc134823326)

[Hình 1.4: Triển khai Static block initialization 5](#_Toc134823327)

[Hình 1.5: Triển khai Thread Safe Singleton 5](#_Toc134823328)

[Hình 1.6: Triển khai Double Check Locking Singleton 6](#_Toc134823329)

[Hình 1.7: Triển khai Bill Pugh Singleton Implementation 7](#_Toc134823330)

[Hình 1.8: Triển khai Enum Singleton 7](#_Toc134823331)

[Hình 1.9: Class DatabaseManager sử dụng Singleton 9](#_Toc134823332)

[Hình 1.10: Tạo 2 biến connection 10](#_Toc134823333)

[Hình 1.11: Kết quả hai biến connection là giống nhau 10](#_Toc134823334)

[Hình 1.12: Class Retrofit Client 11](#_Toc134823335)

[Hình 1.13: Class Activity 12](#_Toc134823336)

[Hình 1.14: Hai đối tượng Retrofit là như nhau 12](#_Toc134823337)

[Hình 1.15: Hình ảnh về máy in 13](#_Toc134823338)

[Hình 1.16: Class employee 14](#_Toc134823339)

[Hình 1.17: Thực thi Singleton 14](#_Toc134823340)

[Hình 1.18: Hàm main khởi tạo Employee 15](#_Toc134823341)

[Hình 1.19: Kết quả demo về máy in sử dụng Singleton 15](#_Toc134823342)

[Hình 1.20: Class Logger 16](#_Toc134823343)

[Hình 1.21: Hàm main khởi tạo Logger 17](#_Toc134823344)

[Hình 1.22: Hai đối tượng Logger là giống nhau 18](#_Toc134823345)

[Hình 1.23: Nội dung log được khi vào cùng một file 18](#_Toc134823346)

[Hình 2.1: Class House 19](#_Toc134823347)

[Hình 2.2: Tạo ra các đối tượng House 20](#_Toc134823348)

[Hình 2.3: HouseBuilder 20](#_Toc134823349)

[Hình 2.4: So sánh 2 cách tạo khác nhau 21](#_Toc134823350)

[Hình 2.5: Kiến trúc của Builder Design Pattern 22](#_Toc134823351)

[Hình 2.6: Class Car 24](#_Toc134823352)

[Hình 2.7: Interface Buider thiết lập các thuộc tính của Car 25](#_Toc134823353)

[Hình 2.8: Class CarBuilder 26](#_Toc134823354)

[Hình 2.9: Class Director chứa các phương thức để tạo đối tượng 27](#_Toc134823355)

[Hình 2.10: Hàm main khởi chạy tạo Car 28](#_Toc134823356)

[Hình 2.11: Tạo Object thành công từ Builder 28](#_Toc134823357)

[Hình 2. 12: Đối tượng House (Product) 29](#_Toc134823358)

[Hình 2.13: Interface Buider triển khai các thành phần của House 29](#_Toc134823359)

[Hình 2.14: Class Director gọi tới Builder để tạo đối tương House 30](#_Toc134823360)

[Hình 2.15: Tạo thành công Object từ House Builder 31](#_Toc134823361)

[Hình 2.16: Class cake 32](#_Toc134823362)

[Hình 2.17: Class muffin 32](#_Toc134823363)

[Hình 2.18: Class pizza 33](#_Toc134823364)

[Hình 2.19: Class Custard 33](#_Toc134823365)

[Hình 2.20: Class demo 34](#_Toc134823366)

[Hình 2.21: Kết quả thực hiện đã tạo Object thành công 34](#_Toc134823367)

[Hình 2.22: Thành phần Product (Airline Ticket) 35](#_Toc134823368)

[Hình 2.23: Thành phần Builder (AirlineTickerBuilder) 36](#_Toc134823369)

[Hình 2.24: Thành phần ConcreteBuilder (AirlineTicketBuilderImpl) 36](#_Toc134823370)

[Hình 2.25: Thành phần Director (AirlineTicketCenter) 37](#_Toc134823371)

[Hình 2.26: Tạo thành công một đối tượng bằng Builder Design Pattern 37](#_Toc134823372)

[Hình 3.1: Class Diagram 39](#_Toc134826719)

[Hình 3.2: So sánh 2 cách triển khai 40](#_Toc134826720)

[Hình 3.3: Sơ đồ Diagram của Façade Design Pattern 41](#_Toc134826721)

[Hình 3.4: Class Amplifier 44](#_Toc134826722)

[Hình 3.5: Class Lights 44](#_Toc134826723)

[Hình 3.6: Class Projector 45](#_Toc134826724)

[Hình 3.7: Class PopcornPopper 45](#_Toc134826725)

[Hình 3.8: Class Screen 45](#_Toc134826726)

[Hình 3.9: Class DvdPlayer 46](#_Toc134826727)

[Hình 3.10: Class Facade trong hệ thống rap phim 47](#_Toc134826728)

[Hình 3.11: Thêm các phương thức watchMovie và endMovie 48](#_Toc134826729)

[Hình 3.12: Hàm main khởi tạo homeTheater 48](#_Toc134826730)

[Hình 3.13: Kết quả đạt được 49](#_Toc134826731)

# **NỘI DUNG**

## **I. Singleton Design Pattern(“Duy nhất”)**

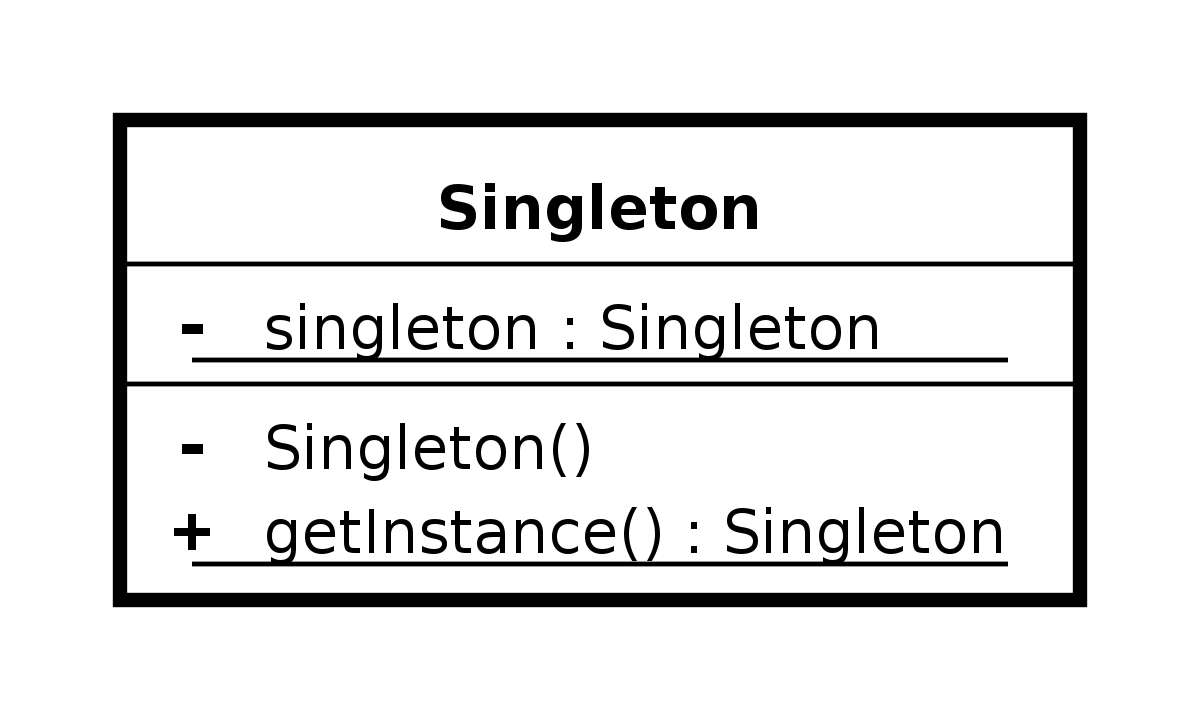
* 1. **Giới thiệu về Singleton Design Pattern**
* Singleton Design Pattern là mẫu thiết kế thuộc loại cấu trúc Creational Pattern, được sử dụng để đảm bảo rằng chỉ có một thể hiện (instance) duy nhất của lớp (class) được tạo và sử dụng trong toàn bộ chương trình. Đồng thời Singleton cung cấp một điểm truy cập toàn cục, có nghĩa là bất kỳ đâu trong chương trình đều có thể truy cập vào nó được.
* Singleton Design Pattern sài được tốt nhất trong những trường hợp sau:

+ Logging: Lưu lại lịch sử người dùng, hành vi người dùng (cái này chúng ta có thể sử dụng Singleton Design Pattern bởi vì chúng ta chỉ cần một Object Log thôi, chỉ bất kỳ đâu, và có thể lặp đi lặp lại nhiều lần).

+ Config File: Khi chúng ta khởi chạy một ứng dụng, có những Config chúng ta sẽ lưu nó vào một biến Singleton. Đảm bảo rằng chỉ config ở một chổ.

+ Kết nối database: Khi triển khai kết nối đến cơ sở dữ liệu, việc sử dụng Singleton Design Pattern để đảm bảo rằng chỉ có một kết nối đến cơ sở dữ liệu trong toàn bộ ứng dụng.

* Trong các trường hợp trên nếu chúng ta khởi tạo nhiều hơn một đối tượng chúng ta sẽ gặp phải các vấn đề về hành vi không chính xác, chiếm dụng tài nguyên. Khi triển khai Singleton sẽ giúp đảm bảo tính nhất quán và đồng bộ giữa các hoạt động, đồng thời giảm tác động đến hiệu suất do việc tạo ra đối tương trong chương trình.
  1. **Sơ đồ UML của Singleton Design Pattern**



Hình 1.1: Kiến trúc của Singleton Design Pattern

* Singleton Pattern có sơ đồ lớp đơn giản nhất, trong thực tế Singleton chỉ nằm trong một lớp duy nhất, mặc dù Singleton đơn giản trong thiết kế lớp nhưng khi triển khai chúng ta vẫn gặp một vài vấn đề nhỏ khi áp dụng nó.
* Lớp Singleton là một lớp tự quản lý một thể hiện (instance) của nó. Chúng ta không cho phép bất kỳ lớp nào khác tự khởi tạo một Instance mới từ nó. Để có được một instance, bạn phải gọi thông qua chính lớp đó.
* Thành phần trong lớp Singleton:

+ Construct có phạm vi truy cập là private để đảm bảo rằng các class bên ngoài không khởi tạo đối tượng từ class này thông qua từ khóa new, chỉ bên trong class Singleton mới có quyền khởi tạo đối tượng của class này.

+ Một biến static để nắm giữ một instance của class Singleton.

+ Một method public static getInstance() trả về một đối tượng của class Singleton. Đây là điểm truy cập toàn cục cho instance, bất kỳ khi nào cần instance ta chỉ cần truy cập và nó sẽ return lại cho bạn một instance duy nhất.

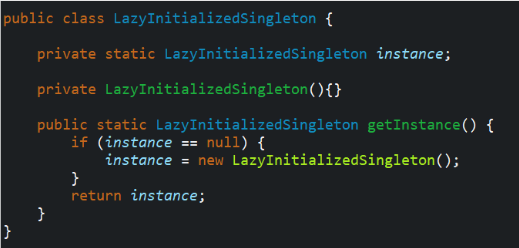
**Câu hỏi đặt ra là tại sao chúng ta không sử dụng biến toàn cục (global variable) thay vì Singleton?**

* Có thể sử dụng Global variable để đảm bảo truy cập vào một instance và truy cập toàn cục trong chương trình, trong Java hoặc các ngôn ngữ lập trình khác có khái niệm static để giúp ta làm điều đó tuy nhiên khi làm điều này sẽ có vài nhược điểm:

+ Khi gán một đối tượng cho một biến toàn cục thì đối tượng đó có thể được tạo khi ứng dụng vừa bắt đầu và chương trình không kết thúc việc sử dụng nó vì vậy sẽ gây ra vấn đề lãng phí tài nguyên.

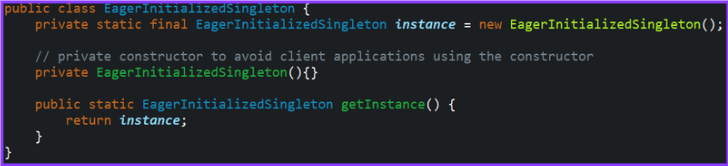
+ Một biến toàn cục cung cấp quyền truy cập global, nhưng không đảm bảo chỉ có một thể hiện của lớp tồn tại.

* 1. **Triển khai Singleton Design Pattern**
     1. **Lazy Initialized Singleton**
* Khi sử dụng cách triển khai này, một thể hiện (instance) sẽ được khởi tạo khi nó được yêu cầu lần đầu tiên.



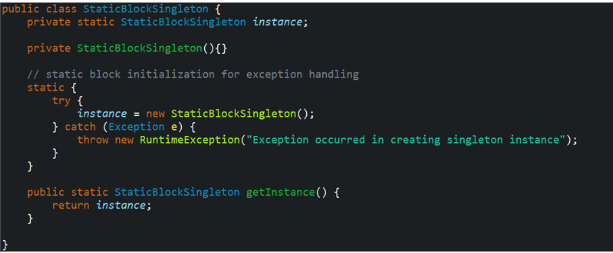
Hình 1.2: Triển khai Lazy Initialized

* Trong cách này biến instance được khai báo là static và không được khởi tạo ngay khi lớp được tải vào bộ nhớ. Thay vào đó instance được khởi tạo khi phương thức getInstance() được gọi lần đầu tiên.
* Tuy nhiên khi triển khai cách này sẽ không đảm bảo tính đồng bộ (thread-safety) và có thể dẫn đến việc tạo ra nhiều thể hiện (instance). Đó là khi instance chưa được khởi tạo và có nhiều thread cùng truy cập vào hàm getInstance() cùng lúc. Lúc này giá trị instance đang là null và nhiều thread cùng vào khối lệnh If và thực hiện việc **new** đối tượng sẽ dẫn đến tạo ra nhiều instance trên các vùng nhớ khác nhau. Điều này là vi phạm tính chất Singleton (chỉ đảm bảo một instance của class trong chương trình).
* Vì vậy cách này chỉ phù hợp khi triển khai trong một chương trình không chạy bất đồng bộ. Khi chương trình chạy bất đồng bộ ta phải sử dụng cách triển khai khác như Double Checked Locking, Enum Singleton, hoặc Eager Initialized Singleton
  + 1. **Eager Initialized Singleton**
* Khi sử dụng cách này, một instance của lớp sẽ được tạo ra ngay khi lớp được tải vào bộ nhớ



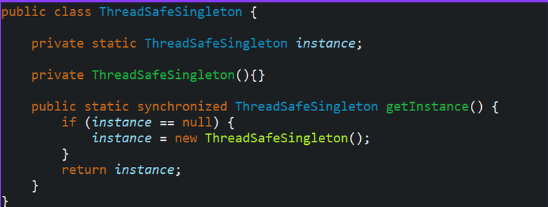
Hình 1.3: Triển khai Eager Initialized

* Cách triển khai này đảm bảo không bị vấn đề (thread-safety) như cách trên và dễ dàng triển khai, tuy nhiên nó có thể dẫn đến việc tạo ra một thể hiện của lớp ngay cả khi nó không được sử dụng trong ứng dụng. Do đó, nếu không chắc rằng lớp Singleton sẽ được sử dụng trong suốt quá trình thực thi của ứng dụng, ta nên cân nhắc sử dụng cách triển cách triển khai khác.
  + 1. **Static block initialization**
* Các triển khai này tương tự như Eager Initialized Singleton nhưng có thêm một khối static block để handle exception hoặc các xử lí khác.

****

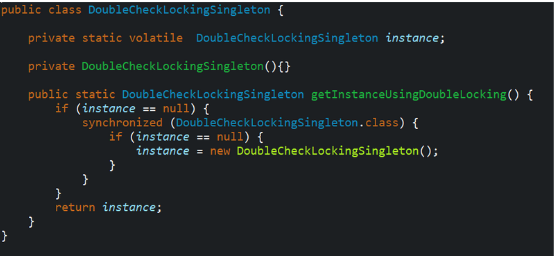
Hình 1.4: Triển khai Static block initialization

* Thay vì khởi tạo instance sẵn thì ta sẽ khởi tạo instace trong khối static block
  + 1. **Thread Safe Singleton**
* Cách triển này sẽ giúp ta sử xữ lí được đa luồng (Thread).

****

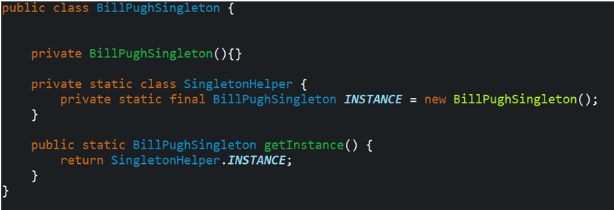
Hình 1.5: Triển khai Thread Safe Singleton

* Ta sẽ gọi phương thức **synchronized** của hàm **getInstance()** và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.
* Biến volatile trong Java có tác dụng thông báo sự thay đổi giá trị của biến tới các thread khác nhau nếu biến này đang được sử dụng trong nhiều thread
* Tuy nhiên nhược điểm của cách triển khai này chính là synchronized. Nó giúp giải quyết được đa luồng nhưng lại chạy rất chậm. Ta có thể thấy rằng chỉ cần synchrozied một lần. Cách triển khai tiếp theo sẽ giải quyết vấn đề này
  + 1. **Double Check Locking Singleton**
* Triển khai theo cách này sẽ đảm bảo thể hiện (instance) của lớp được tạo ra chỉ khi nó được yêu cầu lần đầu tiên. Cách triển khai này đảm bảo tính đồng bộ (thread-safe) và hiệu quả trong việc sử dụng bộ nhớ.
* Cách triển khai này là cách cải tiến của Thread Safe Singleton, khi nó có cho phép chạy bất đồng bộ khi nhiều thread cùng truy cập vào hàm getInstance() cùng một lúc.



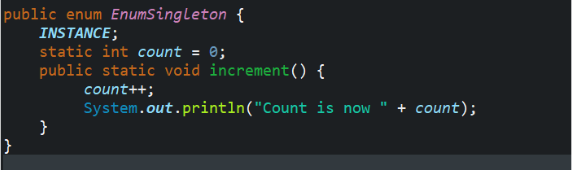
Hình 1.6: Triển khai Double Check Locking Singleton

* Trong phương thức getInstance(), nếu instance chưa được khởi tạo, một khối synchronized được sử dụng để đảm bảo rằng chỉ có một luồng được phép truy cập vào khối lệnh tạo instance. Sau khi luồng đầu tiên tạo ra thể hiện của lớp, các luồng khác sẽ không phải truy cập vào khối synchronized nữa.
  + 1. **Reflection & Bill Pugh Singleton Implementation**
* Reflection là một tính năng trong Java (và cũng có trong một số ngôn ngữ lập trình khác) cho phép các chương trình có thể tương tác, thực thi và kiểm tra thông tin về các đối tượng trong thời gian chạy.
* Reflection cho phép chương trình lấy thông tin về các class, interfaces, các trường (fields), phương thức (methods), các constructor, các annotations và các thông tin khác của các đối tượng trong thời gian chạy. Nó cũng cho phép chương trình tạo mới các đối tượng, gọi các phương thức, thay đổi giá trị của các trường và thực hiện các thao tác khác trên các đối tượng trong thời gian chạy.
* Việc sử dụng Reflection sẽ phá vỡ cấu trúc của Singleton. Ta có cách triển khai Bill Pugh Singleton Implementation sẽ giải quyết vấn đề này
* Ta tạo ra static nested class với vai trò Helper khi muốn tách biệt chức năng cho một class function rõ ràng hơn.



Hình 1.7: Triển khai Bill Pugh Singleton Implementation

* Khi Singleton được tải vào bộ nhớ thì SingletonHelper chưa được tải vào. Nó chỉ được tải khi và chỉ khi phương thức getInstance() được gọi. Với cách này tránh được lỗi cơ chế khởi tạo instance của Singleton trong Multi-Thread, performance cao do tách biệt được quá trình xử lý. Reflection chỉ có thể tấn công và thay đổi class BillPughSingleton trong khi đó instance được khởi tạo trong một static nested class. Do đó, cách làm này được đánh giá là cách triển khai Singleton nhanh và hiệu quả nhất.
  + 1. **Enum Singleton**
* Khi sử dụng Enum thì các tham số được tạo ra là độc nhất vì vậy nó đảm bảo được tính chất của Singleton

****

Hình 1.8: Triển khai Enum Singleton

Trên đây là các cách để triển khai Singleton Tùy thuộc vào yêu cầu của dự án và mục đích sử dụng của lớp Singleton ta sẽ triển khai Singleton một cách phù hợp.

* 1. **Ưu điểm và nhược điểm của Singleton Design Pattern**
* **Nhược điểm:**

+ Triển khai singleton không thể phân lớp (subclass) code Singleton của mình, vì constructor trong lớp Singleton là private, ta không thể mở rộng một lớp với một constructor là private.

+ Khó phát hiện lỗi: Việc sử dụng Singleton có thể dẫn đến các vấn đề về đồng bộ và tính nhất quán trong quá trình sử dụng Singleton. Tuy nhiên, lỗi này khó phát hiện và khó khăn trong việc sửa chữa.

+ Quy pham đến nguyên tắc “Single Responsibility Principle”, Singleton nó không chỉ chịu trách nhiệm quản lý việc tạo 1 instance của nó mà còn chịu trách nhiệm cho bất kỳ vấn đề nào trong ứng dụng như: kết nối DB, đăng ký registry, ghi log…

+ Khó tái sử dụng: Singleton không thể tái sử dụng trong các ứng dụng khác mà có các yêu cầu khác nhau. Điều này có thể gây ra các vấn đề về bảo trì và mở rộng khi các yêu cầu thay đổi.

* **Ưu điểm:**

+ Cung cấp một thể hiện (instance) duy nhất của một lớp trong toàn bộ ứng dụng, giảm thiểu sự trùng lặp và sử dụng bộ nhớ.

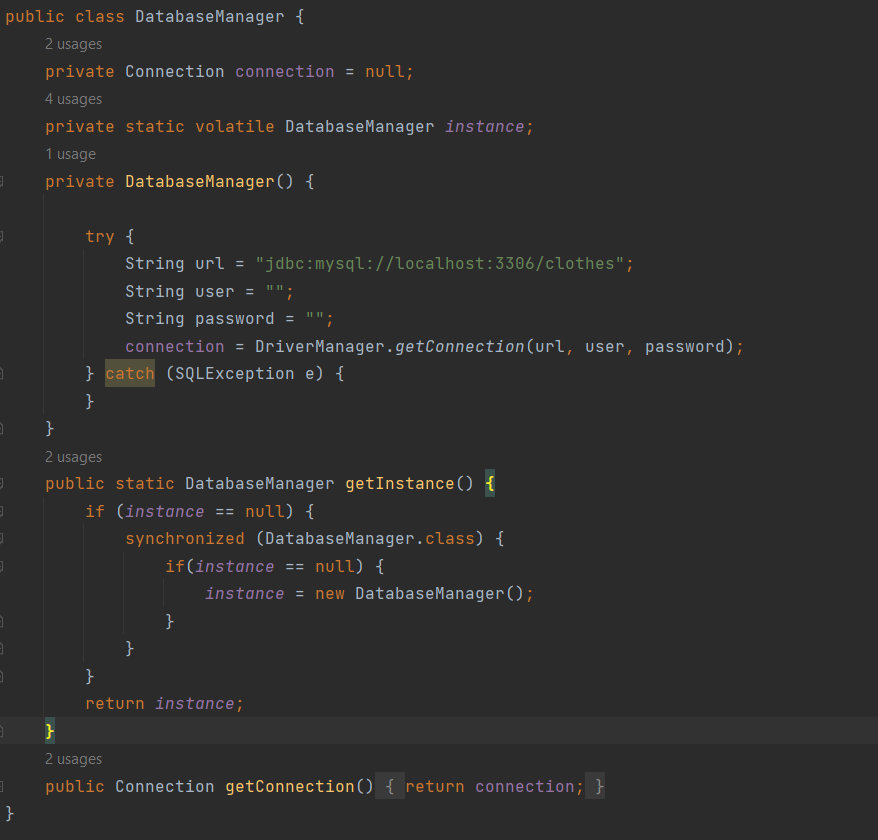
+ Đảm bảo tính đồng bộ trong việc truy cập và sử dụng instance của lớp, ngăn chặn việc tạo ra nhiều thể hiện (instance) của lớp trong ứng dụng.

+ Giúp tăng tính linh hoạt và tiện lợi trong việc sử dụng các đối tượng của lớp Singleton, giảm thiểu sự phức tạp trong việc quản lý các đối tượng

* 1. **Demo**

#### **Phạm Ngọc Thắng**

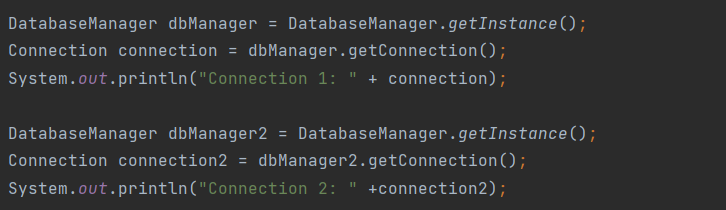
Sử dụng Singleton Design Pattern để quản lý kết nối tới cơ sở dữ liệu.



Hình 1.9: Class DatabaseManager sử dụng Singleton

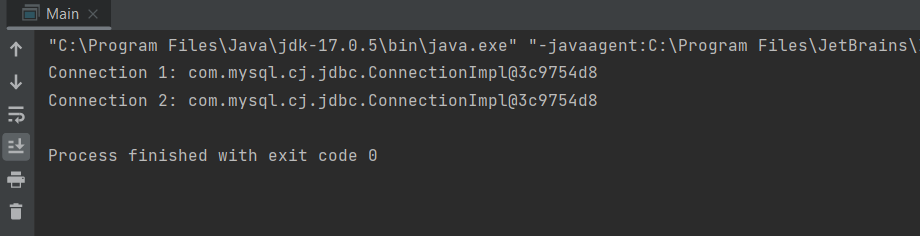
Lớp DatabaseManager triển khai một mẫu thiết kế singleton với phương thức getInstance(), đảm bảo rằng chỉ có một thể hiện của lớp được tạo ra và sử dụng trong toàn bộ ứng dụng.Phương thức khởi tạo của lớp này tạo kết nối tới cơ sở dữ liệu MySQL định sẵn với thông tin đăng nhập. Phương thức getConnection() trả về kết nối đã được khởi tạo. Lớp này sử dụng JDBC để kết nối tới cơ sở dữ liệu MySQL và nó có thể được sử dụng trong các ứng dụng Java để tương tác với cơ sở dữ liệu.

Trong hàm main em sẽ khái báo 2 biến kiểu DatabaseManager đều có giá trị khởi tạo được trả về từ hàm DatabaseManager.getInstance() sau đó em gọi lại hàm getConnection để lấy kết nối đến cơ sở dữ liệu.



Hình 1.10: Tạo 2 biến connection

Sau đó em sẽ in ra thử 2 biến connection



Hình 1.11: Kết quả hai biến connection là giống nhau

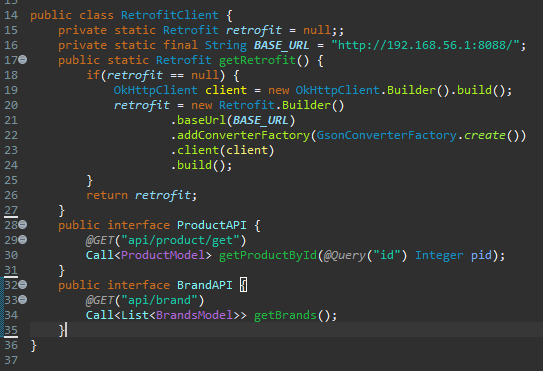
Khi chạy chương trình em thấy rằng 2 connection này đều giống nhau đều tham chiếu đến cùng một địa chỉ vùng nhớ. Em đã triển khai Singleton trên class DatabaseManager thành công. Bây giờ ở bất kỳ đâu trong chương trình muốn tạo Instance từ class đều gọi qua hàm getInstance(), nó sẽ đảm bảo rằng những biến đó đều tham chiếu đến cùng một vùng nhớ.

Nếu ta không triển khai Singleton trên class DatabaseManager thì mỗi lần chúng ta cần sử dụng DatabaseManager ở các class khác nhau thì ta phải tạo mới một đối tượng ở các vùng nhớ khác nhau. Điều này sẽ tiêu tốn tài nguyên và gây ra các vấn đề xử lý đồng thời. Do đó triển khai Singleton là cách tốt nhất để quản lý kết nối tới cơ sở dữ liệu trong chương trình.

#### **Nguyễn Sinh Hùng**

Trong một ứng dụng Android, bạn sẽ cần một instance toàn cục của đối tượng Retrofit để các thành phần khác của ứng dụng chẳng hạn như HomeActivity hoặc ProductActivity, có thể sử dụng nó để thực hiện một kết nối network, mà không cần tạo một instance mỗi lần chúng cần nó.

Thiết kế class RetrofitClient theo Singleton Design Pattern



Hình 1.12: Class Retrofit Client

Bất cứ lúc nào client A gọi RetrofitClient.getClient(), nó sẽ tạo ra một instance nếu nó chưa được tạo, và rồi khi client B gọi phương thức này, nó kiểm tra nếu instance của Retrofit đã tồn tại hay chưa. Nếu đã được tạo rồi, nó trả về instance cho client B thay vì tạo ra một instance mới.

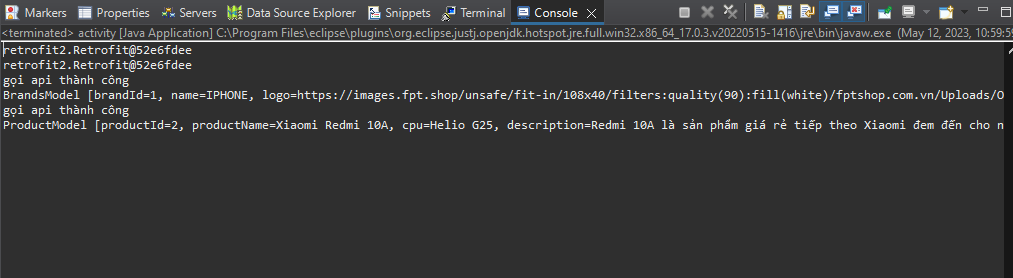
Ngoài ra ta còn có các interface ProductAPI và BrandAPI được sử dụng để định nghĩa các phương thức để truy vấn API.



Hình 1.13: Class Activity

Đoạn code trên là một ví dụ về việc sử dụng RetrofitClient để truy vấn API thông qua hai interface ProductAPI và BrandAPI.

**Kết quả Demo**



Hình 1.14: Hai đối tượng Retrofit là như nhau

Như ta có thể thấy, chúng ta tạo ra hai instance Retrofit thông qua hai interface ProductAPI và BrandAPI nhưng khi in ra màn hình kết quả hai instance Retrofit đều giống nhau.

#### **1.5.3. Vũ Hoàng Anh**

**Demo Singleton Sử dụng trong máy in**: Trong một công ty có 1 máy in, nhiều nhân viên muốn in tài liệu, khi quá nhiều yêu cầu tới cùng một lúc, sẽ xảy ra tình trạng lỗi trong máy không thể xử lý.



Hình 1.15: Hình ảnh về máy in

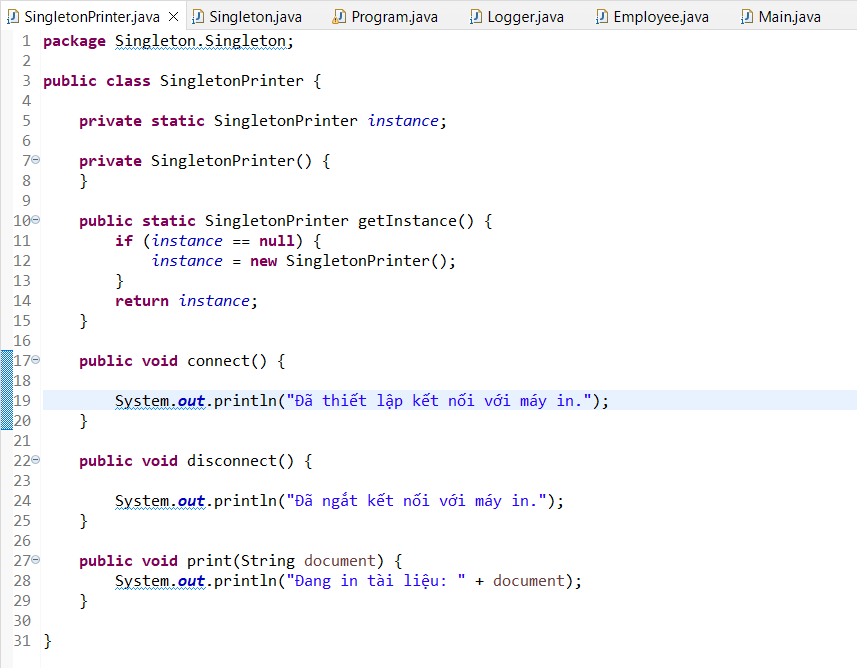
**Class Singleton**

Tạo ra một đối tượng máy in duy nhất, đảm bảo chỉ có một nhân viên chỉ in  duy nhất tại cùng 1 thời điểm

Lớp SingletonPrinter có một biến tĩnh private kiểu SingletonPrinter được gọi là instance, và một phương thức public static getInstance() để truy xuất đến thể hiện SingletonPrinter.

Trong phương thức getInstance(), nếu instance chưa được khởi tạo, nó sẽ tạo một thể hiện mới của SingletonPrinter bằng cách gọi constructor private và gán cho biến instance. Sau đó, phương thức trả về thể hiện đó.

Lớp SingletonPrinter cũng có các phương thức connect(), disconnect() và print(String document) để kết nối, ngắt kết nối và in tài liệu, tương ứng với một máy in thực tế.

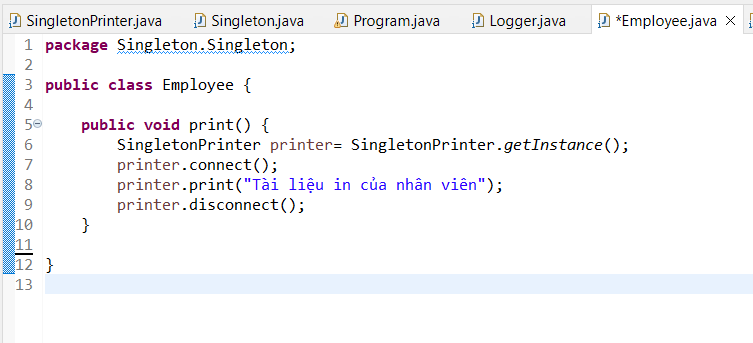


Hình 1.16: Class employee

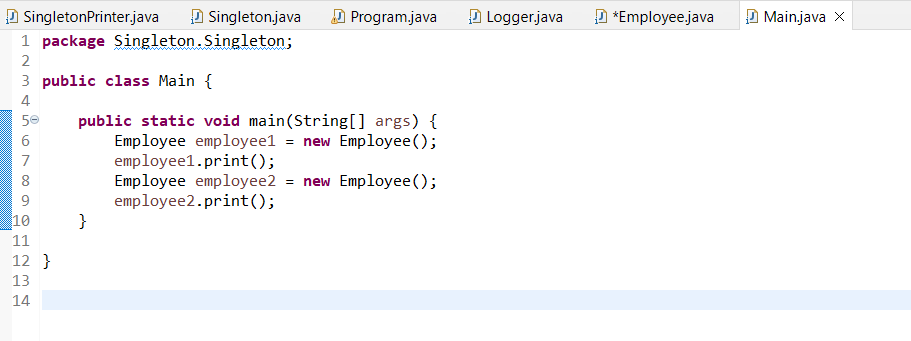
Trong phương thức print() của lớp Employee, đầu tiên nó sử dụng phương thức getInstance() của lớp SingletonPrinter để lấy thể hiện duy nhất của SingletonPrinter.

Sau đó, nó sử dụng đối tượng SingletonPrinter để kết nối với máy in bằng phương thức connect(), in tài liệu bằng phương thức print("Tài liệu in của nhân viên") và ngắt kết nối với máy in bằng phương thức disconnect().

Việc sử dụng mẫu thiết kế Singleton trong đoạn mã này đảm bảo rằng chỉ có một thể hiện của SingletonPrinter được tạo ra và sử dụng trong toàn bộ ứng dụng. Điều này giúp giảm thiểu việc sử dụng bộ nhớ và đảm bảo tính nhất quán của dữ liệu được in ra.

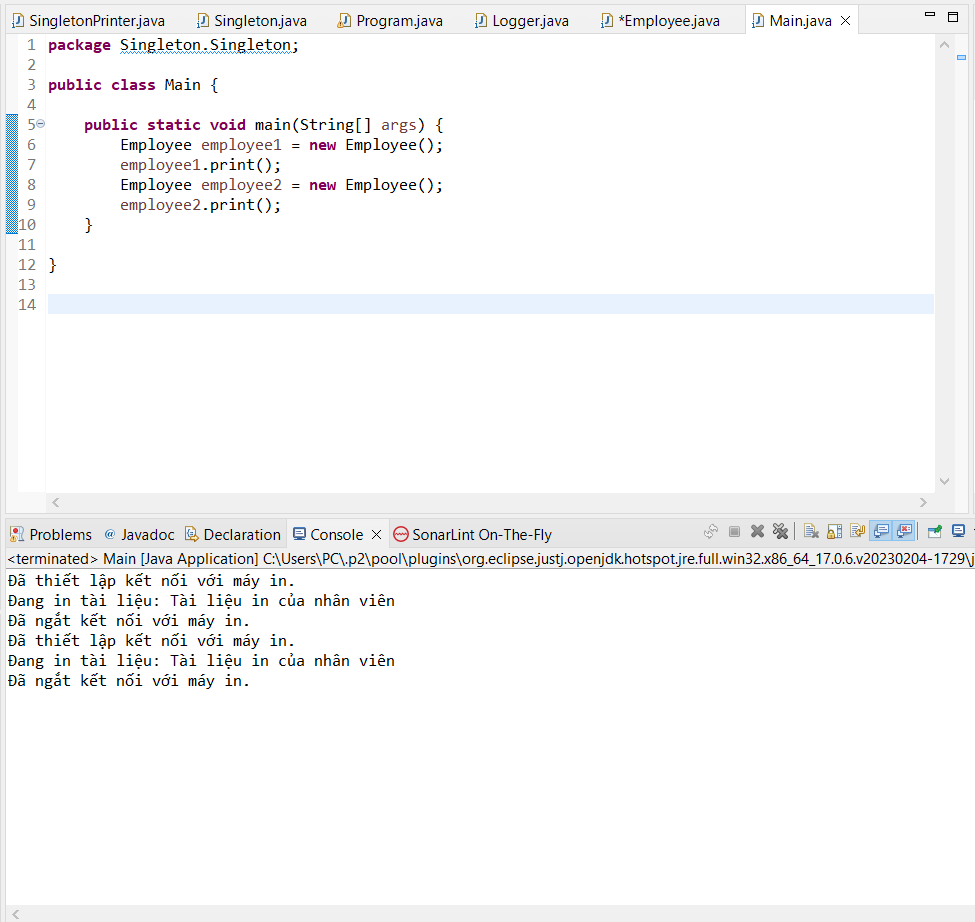


Hình 1.17: Thực thi Singleton



Hình 1.18: Hàm main khởi tạo Employee

**Kết quả demo**



Hình 1.19: Kết quả demo về máy in sử dụng Singleton

#### **1.5.4. Trần Chí Mỹ**

**Khái quát DEMO sử dụng Singleton Design Pattern để ghi log trong ứng dụng.**

Trường hợp của logger, việc sử dụng singleton giúp đảm bảo rằng chỉ có một instance của logger được sử dụng trong toàn bộ ứng dụng. Điều này giúp giảm thiểu tài nguyên và đảm bảo tính nhất quán trong các thông tin được ghi lại. Nếu có nhiều logger instances, chúng có thể gây ra sự lộn xộn về thông tin và cũng làm tăng tài nguyên cần thiết cho các hoạt động ghi lại.

Thiết kế class logger theo Singleton Design Pattern

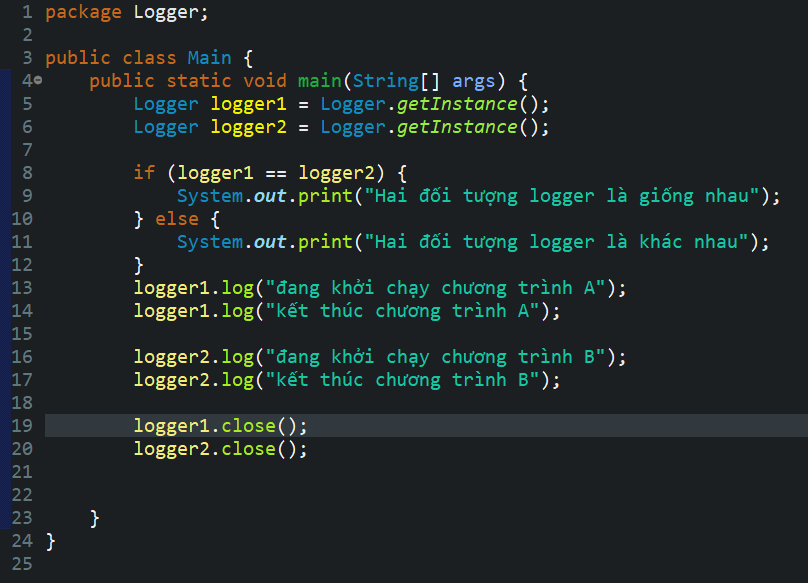


Hình 1.20: Class Logger

Trong class Logger trên có đầy đủ 3 thành phần cần thiết của Singleton Design Pattern

* Private static instance: Đây là instance duy nhất của Logger. Nó được khai báo là private static để đảm bảo rằng nó chỉ có thể được truy cập bởi phương thức getInstance() và không thể được truy cập từ các lớp bên ngoài. Khi được tạo ra, instance này được sử dụng để ghi log vào file "log.txt" thông qua phương thức log().
* Private constructor: Đảm bảo rằng logger chỉ có thể được khởi tạo bên trong lớp Logger. Việc khai báo constructor là private đảm bảo rằng không ai có thể khởi tạo logger từ bên ngoài lớp.
* Static getInstance() method: Đây là phương thức trả về instance duy nhất của Logger. Phương thức này là static, nghĩa là nó thuộc về lớp Logger chứ không phải của một instance cụ thể. Phương thức này sử dụng kiểm tra null để đảm bảo rằng chỉ có một instance được tạo ra và trả về cho các lần gọi tiếp theo.
* Đây là cách triển khai Lazy Initialized Singleton
* Ngoài ra trong class còn có một số hàm để ghi vào file log và đóng file log đó.

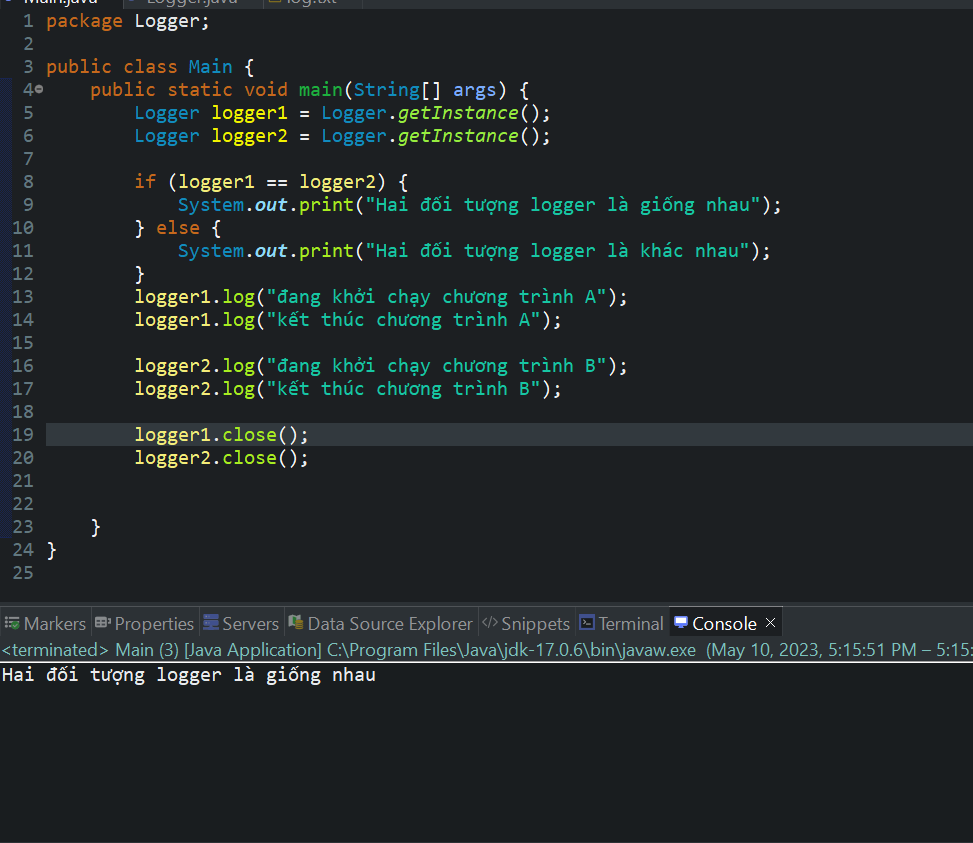
**Tiến hành khởi chạy**



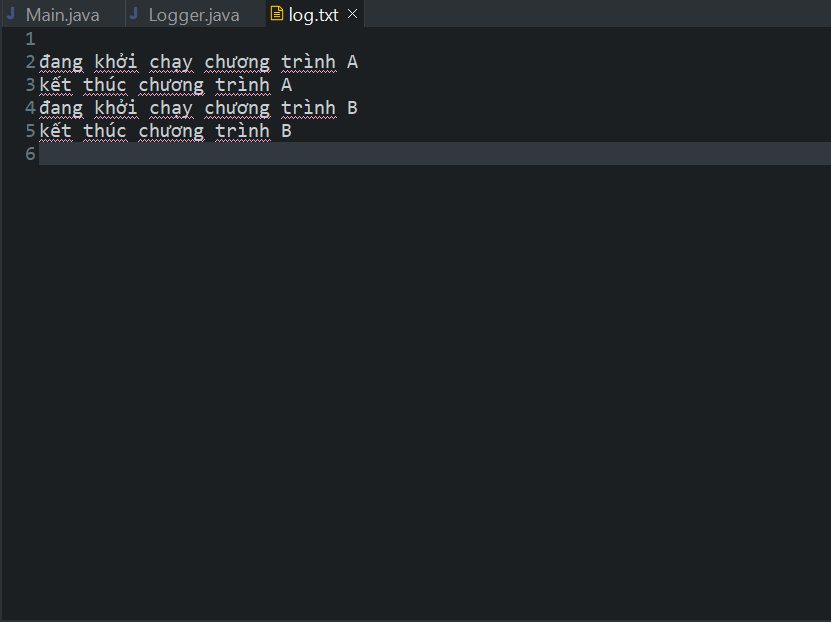
Hình 1.21: Hàm main khởi tạo Logger

Hàm main sẽ tạo 2 đối tượng logger1 và logger2 và kiểm tra xem rằng 2 object này có phải là một hay không. Tiến hành ghi log bằng hai đối tương đó

**Kết quả**

****

Hình 1.22: Hai đối tượng Logger là giống nhau

****

Hình 1.23: Nội dung log được khi vào cùng một file

## **II** **Builder Design Pattern(“Thợ Xây”)**

1. **Giới thiệu về Builder Design Pattern**

* Builder Design Pattern là một trong những design pattern nằm trong loại Creational Pattern – những mẫu thiết kế cho việc khởi tạo đối tượng của lớp
* Builder Design Pattern được tạo ra để xây dựng các đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản, sử dụng cách tiếp cận theo từng bước
* Khi sử dụng Builder Design Pattern thì việc tạo đối tượng sẽ trở nên linh hoạt và dễ dàng, đồng thời làm code chúng ta dễ đọc, dễ bảo trì.

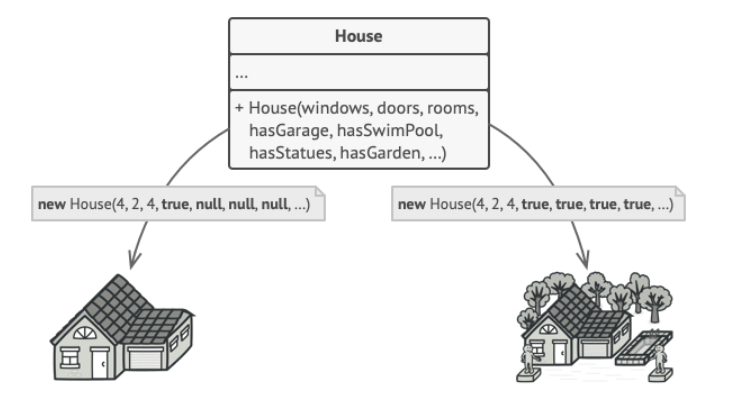
1. **Đặt vấn đề**

* Ở đây ta có một class House có các thuộc tính cơ bản của ngôi nhà



Hình 2.1: Class House

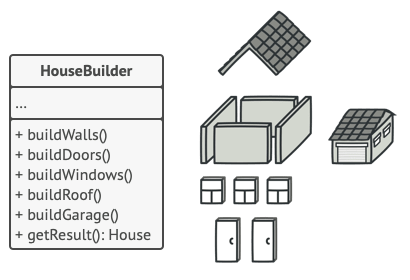
* Và em muốn tạo ra các đối tượng có các thuộc tính khác nhau ví dụ House 1 chỉ có 4 windows, có 2 door, có 4 room, có garage. House 2 giống house 1 nhưng có thêm Swim, Garden. Thông thương khi chúng ta chưa tiếp cần với Builder Design Pattern chúng ta thường khởi tạo đối tượng theo cách sử dụng constructor.



Hình 2.2: Tạo ra các đối tượng House

* Các thuộc tính nào không có giá trị thì để giá trị mặc định hoặc null tuy nhiên khi làm như vậy thì sẽ làm code chúng ta khó đọc. Và khi chúng ta có nhiều trường hợp khởi tạo khác nhau thì ta phải overload constructor nhiều. Nếu class house có nhiều thuộc tính thì rất nhiều constructor trong class và độ dài của constructor lớn.
* Vì vậy giải pháp đưa ra là chúng ta sẽ sử dụng **Builder Design Pattern**

1. **Đặc điểm của Builder Design Pattern**

* Builder Design Pattern sẽ giúp ta chia nhỏ constructor ra làm nhiều phần và cho chúng ta khả năng khởi tạo ra một Object bằng nhiều hướng khác nhau.
* ****

Hình 2.3: HouseBuilder

* Khi khởi tạo đối tượng sử dụng Builder Design Pattern thì việc xây dựng đối tượng sẽ gồm tập hợp các bước (VD như hình trên BuildWall, BuildDoor, BuildRoof,…) và bạn phải thực hiện một loạt các bước này để tạo ra một đối tượng. Nhưng điều quan trong là bạn không nhất thiết phải làm hết tất cả các bước. Ta chỉ cần gọi các bước cần thiết để tạo ra một đối tượng ví dụ như một căn nhà cơ bản ta chỉ cần làm các bước build roof, wall, door. Là có thể tạo ra một ngôi nhà cơ bản rồi.
* Chúng ta sẽ so sánh cách khởi tạo với Builder Design Pattern và cách khởi tạo Constructor thông thường:



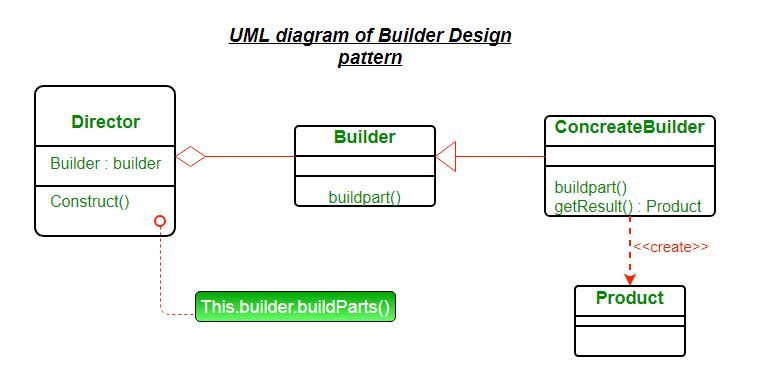
Hình 2.4: So sánh 2 cách tạo khác nhau

* Nhìn vào chúng ta sẽ có cảm giác là sử dụng Builder nó sẽ dài hơn, tuy nhiên nếu chúng ta nhìn nhận và độ dễ đọc thì rõ ràng cách tiếp cận sử dụng Builder Design Pattern nó sẽ dễ đọc hơn rất nhiều, nhìn vào code chúng ta có thể biết được giá trị khởi tạo cho các thuộc tính roof, windows, doors,… là có giá trị gì và chúng có thể thay đổi thứ tự hàm tùy thích. Những thuộc tính nào không yêu cầu chúng ta có thể bỏ ra.
* Một ưu điểm của Builder Design Pattern là chúng ta có thể tạo ra các quy trình xây dựng khác nhau thực hiện một tập hợp các bước xây dựng khác nhau. Sau đó, bạn có thể sử dụng các trình tạo này để tạo ra một đối tượng mong muốn. Ví dụ như tôi có 3 hàm xây dựng House, hàm thứ 1 xây dựng một ngôi nhà gỗ, hàm thứ hai xây dựng nhà gạch, hàm thứ ba xây dựng một lâu đài.



* Chúng ta có thể trích xuất việc xây dựng này vào một class riêng, đó là Direction (một thành phần trong Builder) và Client gọi ra các hàm xây dựng này để có một đối tượng mà mình cần. Khi làm như vậy thì Client không cần biết chi tiết về quá trình khởi tạo, điều này giúp che dấu quá trình khởi tạo từ phía Client và giúp giảm thiểu sự phụ thuộc của class.

1. **Sơ đồ UML của Builder Design Pattern**

****

Hình 2.5: Kiến trúc của Builder Design Pattern

Builder Desgin Pattern sẽ gồm có 4 thành phần chính:

+ Product: đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính. Trong ví dụ trên đó là class House

+ Builder: Là một abstract class hoặc Interface khai báo các phương thức để khởi tạo đối tượng

+ ConcreteBuilder: kế thừa từ Builder và cài đặt chi tiết cách tạo đối tượng. đồng thời nó cũng cung cấp phương thức để trả về các đối tượng.

+ Director: là nơi sẽ gọi tới Builder để tạo ra đối tượng, là nơi chứa các hàm dựng để tạo ra đối tượng mà client cần như đã trình bày ở trên. Lớp Director không cần thiết phải có. Ta có thể gọi đến Builder trực tiếp từ Client. Tuy nhiên, lớp này là nơi tốt để đặt các quy trình xây dựng khác nhau để có thể sử dụng lại chúng trong chương trình.

1. **Ưu và nhược điểm của Builder Design Pattern**

* **Nhược điểm:**

+ Builder Design Pattern yêu cầu thêm nhiều lớp để hổ trợ việc xây dựng đối tượng

+ Duplicate code khá nhiều: do cần phải copy tất cả các thuộc tính từ class Product sang class Builder

* **Ưu điểm**

+ Tách rời việc xây dựng đối tượng và việc tạo ra đối tượng: Builder cho phép ta tách rời việc xây dựng đối tượng khỏi việc tạo ra đối tượng, giúp cho việc xây dựng đối tượng trở nên dễ dàng hơn.

+ Làm cho code dễ đọc, dễ bảo trì hơn

+ Cung cấp khả năng tái sử dụng: Builder Design Pattern cho phép ta sử dụng các phần xây dựng để tạo ra các đối tượng khác nhau, giảm thiểu sự lặp lại trong mã

1. **Khi nào sử dụng Builder Design Pattern**

* Builder Design Pattern là một mẫu thiết kế mãnh mẽ và linh hoạt, giúp cho việc xây dựng các đối tượng phức tạp trở nên đơn giản, cung cấp khả năng tái sử dụng và giúp mã trở nên dễ đọc và dễ hiệu
* Builder Design Pattern nên sử dụng đối với các lớp có số lượng thuộc tính lớn, còn đối với các class đơn giản (có vài thuộc tính < 5) thì nên sử dụng constructor bình thường cho nhanh.
* Trong hầu hết các ngôn ngữ lập trình đều có framework hổ trợ tạo ra lớp Builder như trong Java chúng ta có thể sử dụng thư viên Lombok và sử dụng annotation @Builder để tạo ra class Builder và sử dụng.

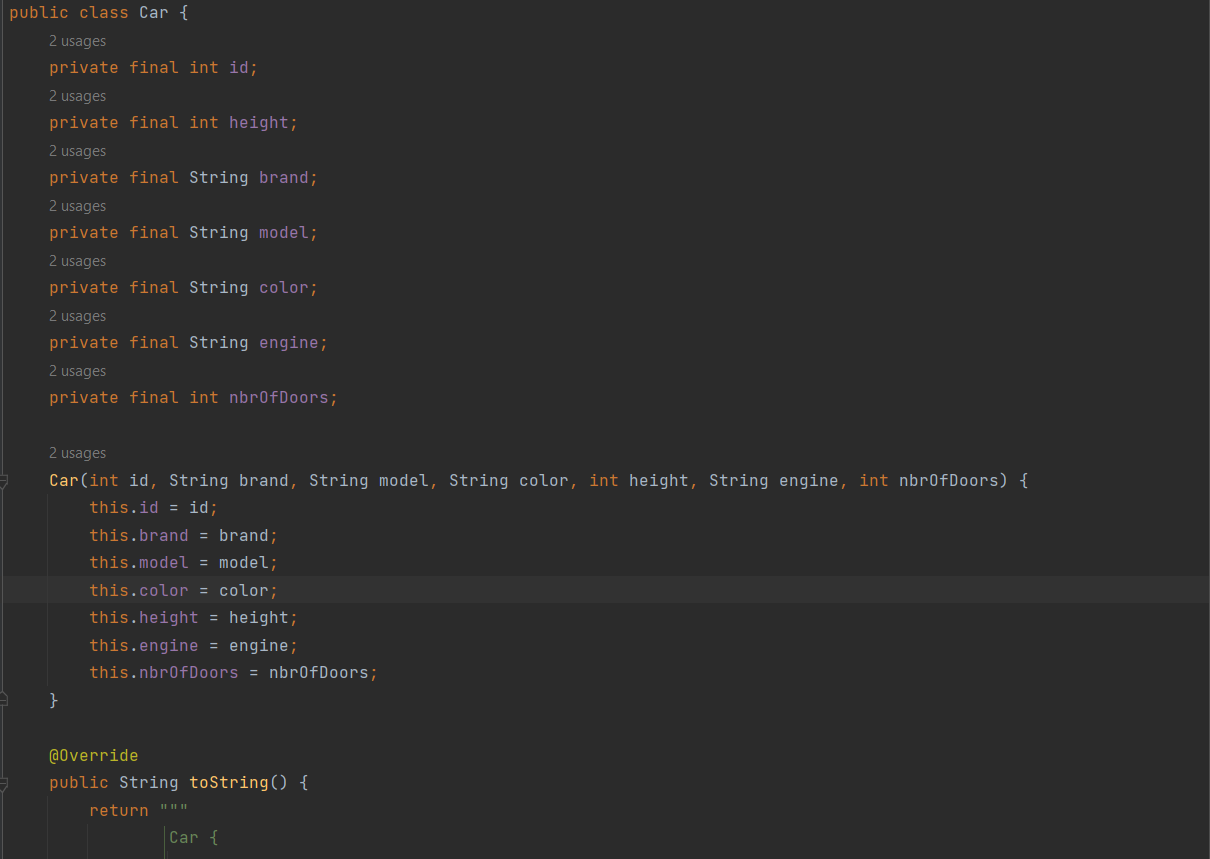
1. **Demo**

#### **2.7.1. Phạm Ngọc Thắng**

Triển khai Builder Design Pattern trên lớp Car

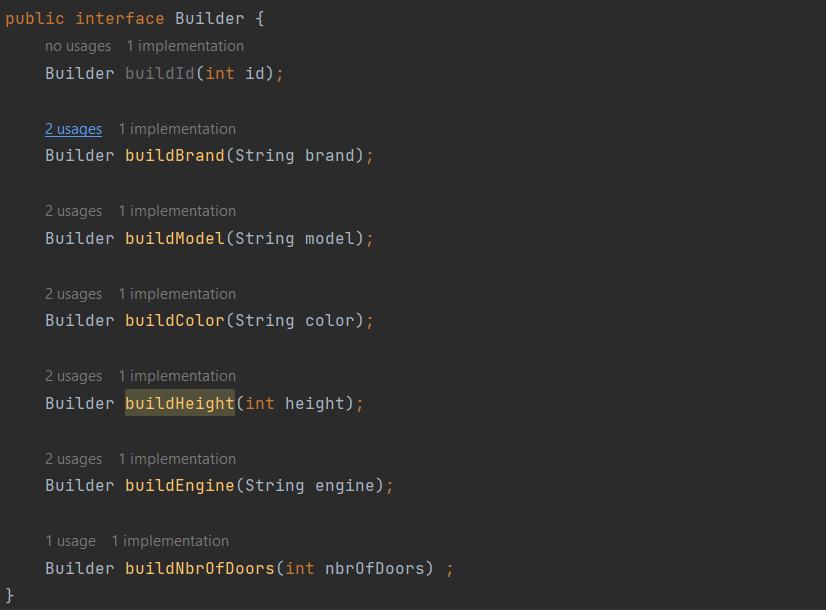
Trong project Demo này em sẽ có bồn thành phần trong Builder Pattern

* Class Car đại diện cho Product trong sơ đồ UML.



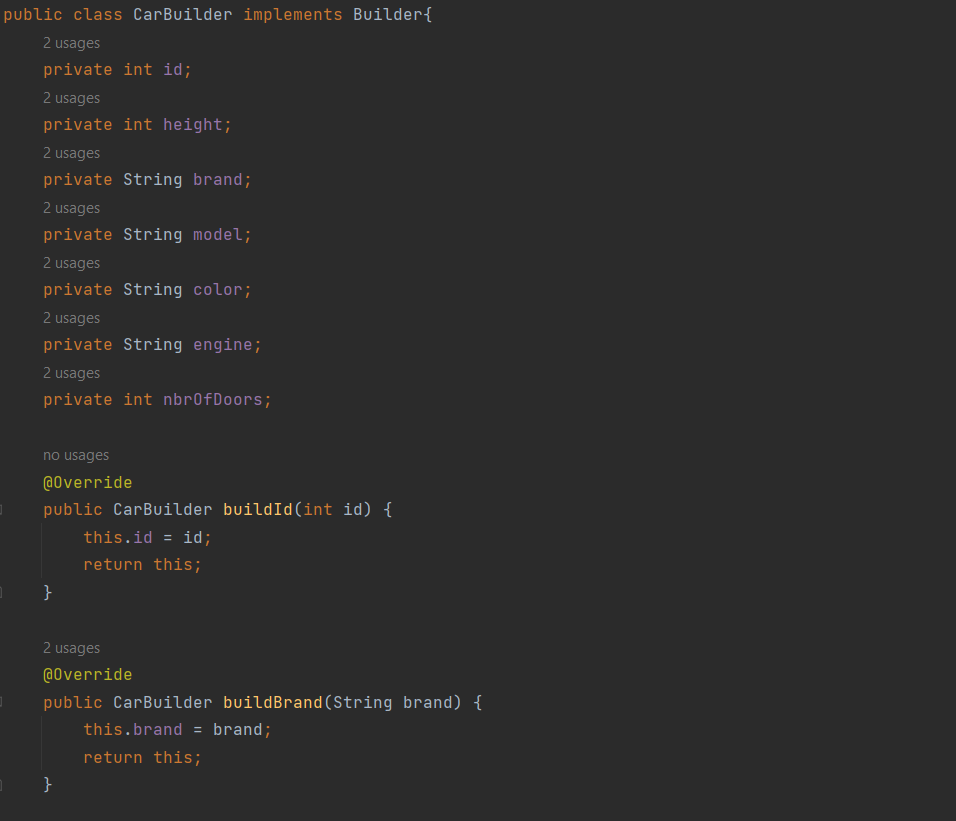
Hình 2.6: Class Car

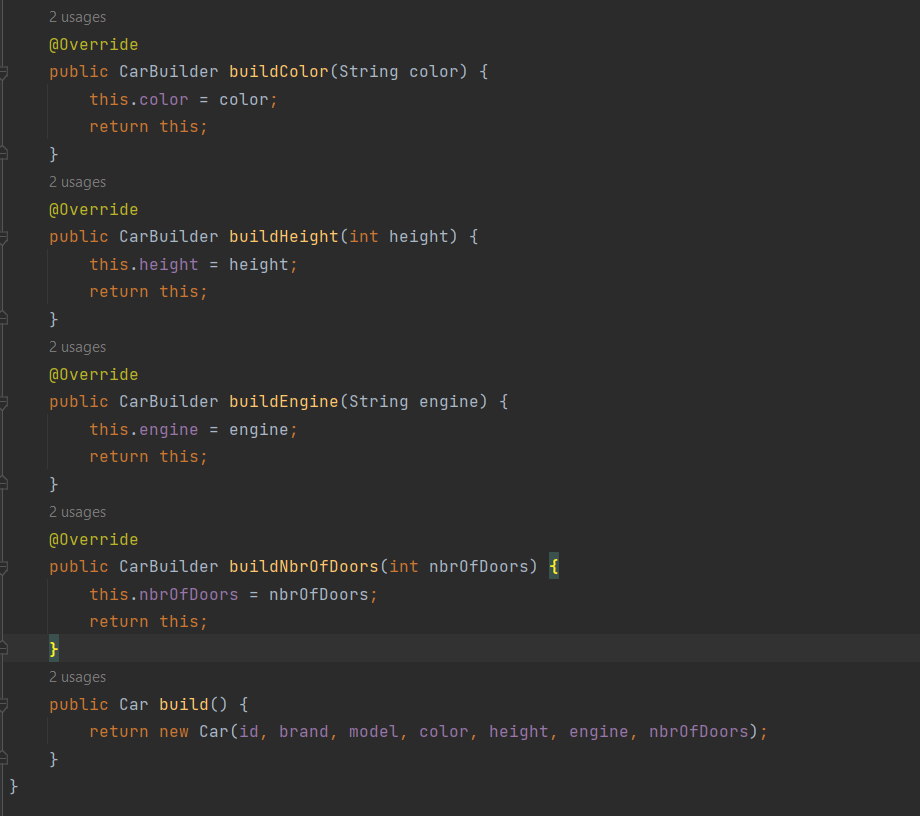
* Interface Builder định nghĩa các phương thức cần thiết để thiết lập các thuộc tính của đối tượng. Những phương thức này cho phép xây dựng đối tượng theo từng bước và chỉ thiết lập các thuộc tính cần thiết.



Hình 2.7: Interface Buider thiết lập các thuộc tính của Car

* Class CarBuilder sẽ duplicate các thuộc tính có trong class Car và  implement các phương thức có trong interface Builder,  sau khi các phương thức để xây dựng các thuộc tính Client sẽ gọi ra phương thức builder() để trả ra đối tượng mà mình cần.





Hình 2.8: Class CarBuilder

* Class Director: Trong này sẽ chứa các phương thức để build các đối tượng mà thường sử dụng trong chương trình, Client gọi các phương thức này là sẽ có đối tượng mình cần VD như gọi hàm buildBugatti thì sẽ có xe Bugatti, gọi buildLambor sẽ có xe Lamborghini.



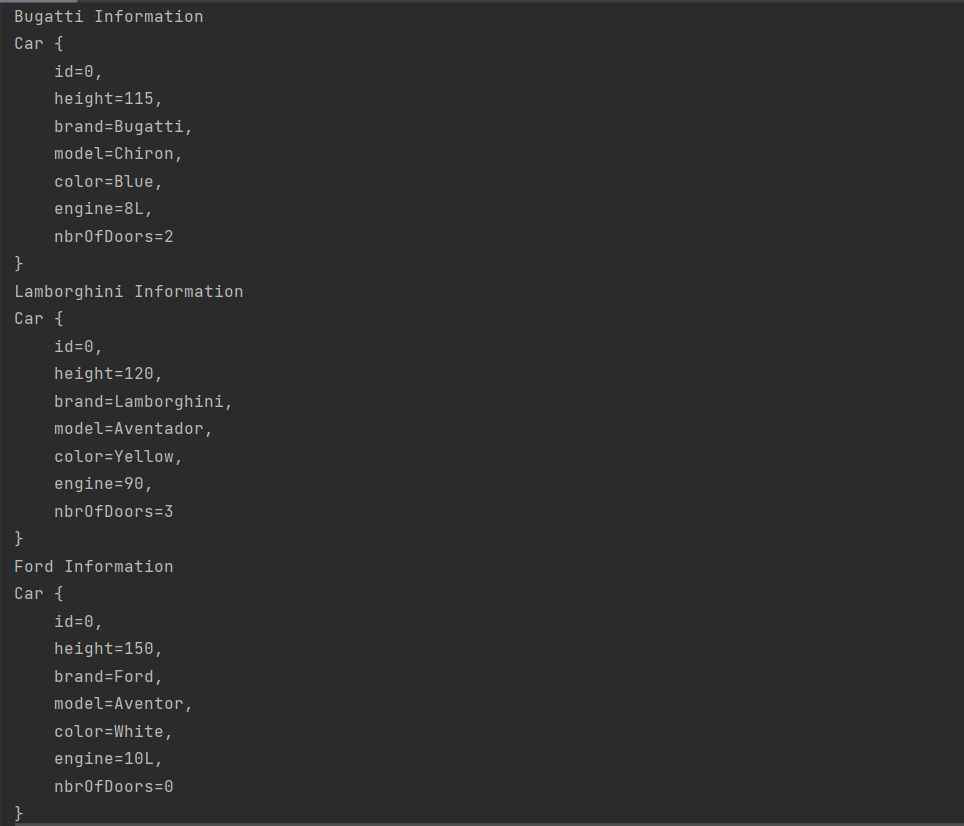
Hình 2.9: Class Director chứa các phương thức để tạo đối tượng

* Trong hàm Main em sẽ gọi phương thức buildBugatti và buildLambo từ Director, sau đó thiết lập lại một vài thuộc tính riêng của đối tượng này bằng cách gọi lại các phương thức build trong Class CarBuilder. Ngoài ra Client có thể khởi tạo trực tiếp đối tượng Car từ CarBuild mà không cần thông qua class Director.



Hình 2.10: Hàm main khởi chạy tạo Car

**Kết quả:**



Hình 2.11: Tạo Object thành công từ Builder

#### **2.7.2. Nguyễn Sinh Hùng**

Ví dụ DEMO tạo Ngôi Nhà (House) sử dụng **Builder Design Pattern**

* Ta sẽ xây dựng một ngôi nhà và tạo ra các đối tượng ngôi nhà với các thuộc tính khác nhau. Các thuộc tính của ngôi nhà như hình bên dưới(class House).

+ ***Đối tượng House (Product):*** Đại diện cho đối tượng cần tạo



Hình 2. 12: Đối tượng House (Product)

+ **Interface Buider :** Khai báo ra các phương thức để khởi tạo đối tượng House



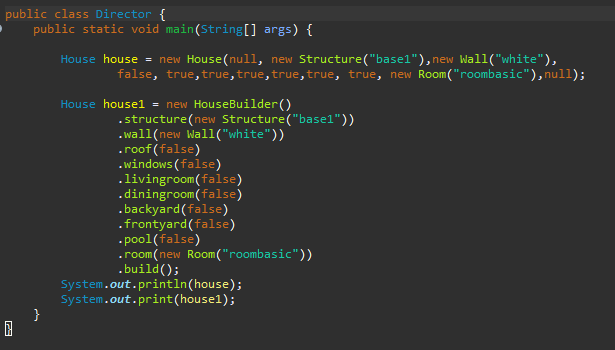
Hình 2.13: Interface Buider triển khai các thành phần của House

**+ HouseBuilder (ConcreteBuilder):** kế thừa từ Builder và cài đặt chi tiết cách tạo ra từng thành phần của đối tượng House



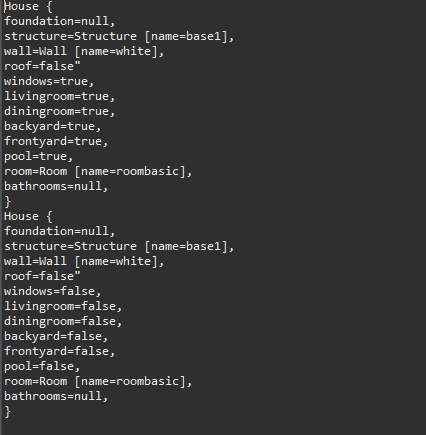
*Class House Builder*

**+ Director:** nơi gọi tới Builder để tạo đối tượng House



Hình 2.14: Class Director gọi tới Builder để tạo đối tương House

**Kết quả Demo:**



Hình 2.15: Tạo thành công Object từ House Builder

* Khởi tạo đối tượng House bằng từ khóa new và gọi bằng Builder đều cho ra kết quả giống nhau nhưng với cách gọi Builder ta thấy nó rõ ràng dễ hiểu hơn, biết được giá trị chính xác khi gọi phương thức và không cần phải truyền giá trị null cho các tham số không sử dụng.

#### **2.7.3. Vũ Hoàng Anh**

**Demo Builder trong việc làm một chiếc bánh**

Lớp Cake có các thuộc tính name, width và height để lưu trữ thông tin về cái bánh. Lớp này có một constructor public để tạo ra đối tượng Cake từ một AbstractBuilder.

Lớp AbstractBuilder là một lớp trừu tượng được sử dụng để triển khai các builder cụ thể cho các đối tượng. AbstractBuilder có các phương thức set cho name, width và height, đồng thời trả về builder hiện tại để có thể sử dụng chuỗi builder. Lớp này cũng có một phương thức trừu tượng getThis() để trả về builder hiện tại, để các lớp builder cụ thể có thể triển khai.



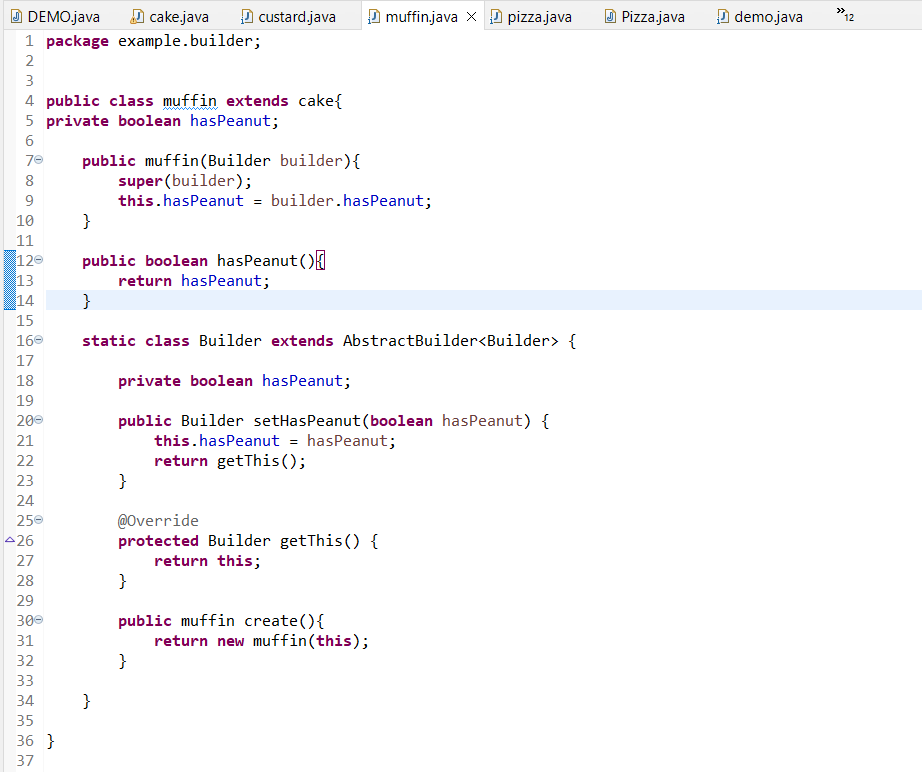
Hình 2.16: Class cake

Lớp Muffin có thêm một thuộc tính boolean hasPeanut để xác định xem món muffin có chứa đậu phộng hay không.

Lớp Muffin có một constructor public để tạo ra đối tượng Muffin từ một Builder cụ thể. Constructor này gọi constructor của lớp cha Cake và gán giá trị cho thuộc tính hasPeanut.

Lớp Builder là một lớp con của lớp AbstractBuilder được sử dụng để triển khai các builder cụ thể cho các đối tượng muffin. Lớp Builder có một phương thức setHasPeanut() để thiết lập thuộc tính hasPeanut và trả về builder hiện tại để có thể sử dụng chuỗi builder. Lớp này cũng triển khai phương thức trừu tượng getThis() để trả về builder hiện tại.

Phương thức create() của lớp Builder được sử dụng để tạo ra một đối tượng Muffin mới từ các giá trị hiện tại của builder.



Hình 2.17: Class muffin

Lớp Pizza có một thuộc tính boolean hasPepper để xác định xem món pizza có chứa hạt tiêu đen hay không.

Lớp Pizza có một constructor public để tạo ra đối tượng Pizza từ một Builder cụ thể. Constructor này gọi constructor của lớp cha Cake và gán giá trị cho thuộc tính hasPepper.

Lớp Builder là một lớp con của lớp AbstractBuilder được sử dụng để triển khai các builder cụ thể cho các đối tượng pizza. Lớp Builder có một phương thức setHasPepper() để thiết lập thuộc tính hasPepper và trả về builder hiện tại để có thể sử dụng chuỗi builder. Lớp này cũng triển khai phương thức trừu tượng getThis() để trả về builder hiện tại.

Phương thức create() của lớp Builder được sử dụng để tạo ra một đối tượng Pizza mới từ các giá trị hiện tại của builder.



Hình 2.18: Class pizza

Lớp Custard có thêm một thuộc tính boolean hasEgg để xác định xem món custard có chứa trứng hay không.

Lớp Custard có một constructor public để tạo ra đối tượng Custard từ một Builder cụ thể. Constructor này gọi constructor của lớp cha Cake và gán giá trị cho thuộc tính hasEgg.

Lớp Builder là một lớp con của lớp AbstractBuilder được sử dụng để triển khai các builder cụ thể cho các đối tượng custard. Lớp Builder có một phương thức setHasEgg() để thiết lập thuộc tính hasEgg và trả về builder hiện tại để có thể sử dụng chuỗi builder. Lớp này cũng triển khai phương thức trừu tượng getThis() để trả về builder hiện tại.

Phương thức create() của lớp Builder được sử dụng để tạo ra một đối tượng Custard mới từ các giá trị hiện tại của builder

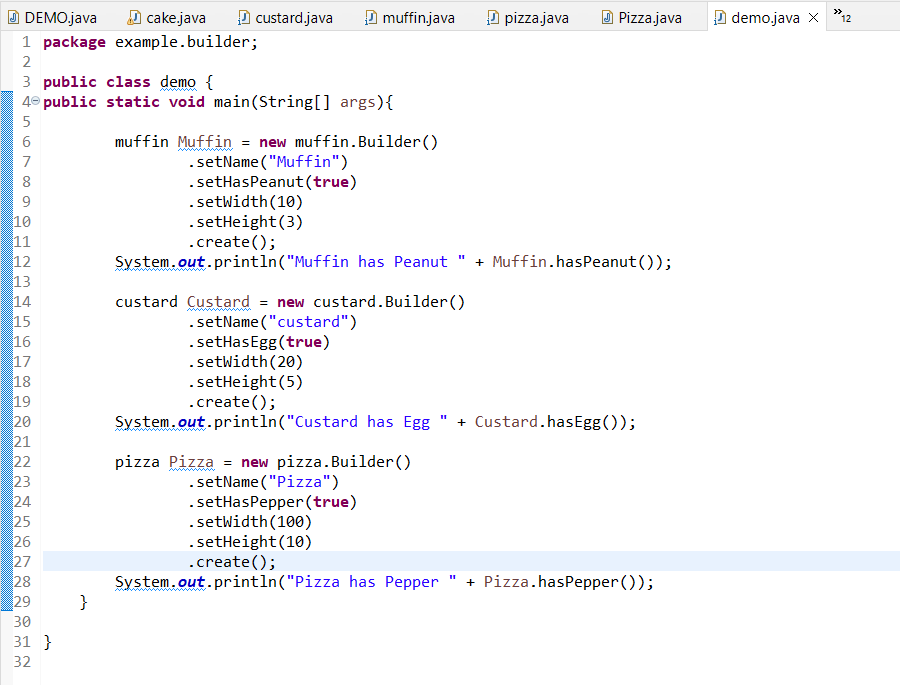


Hình 2.19: Class Custard

Trong phương thức main(), trước tiên tạo một đối tượng Muffin bằng cách sử dụng builder của lớp Muffin để cung cấp các thông tin tạo đối tượng. Sau đó in ra thông tin về việc món muffin có chứa đậu phộng hay không.

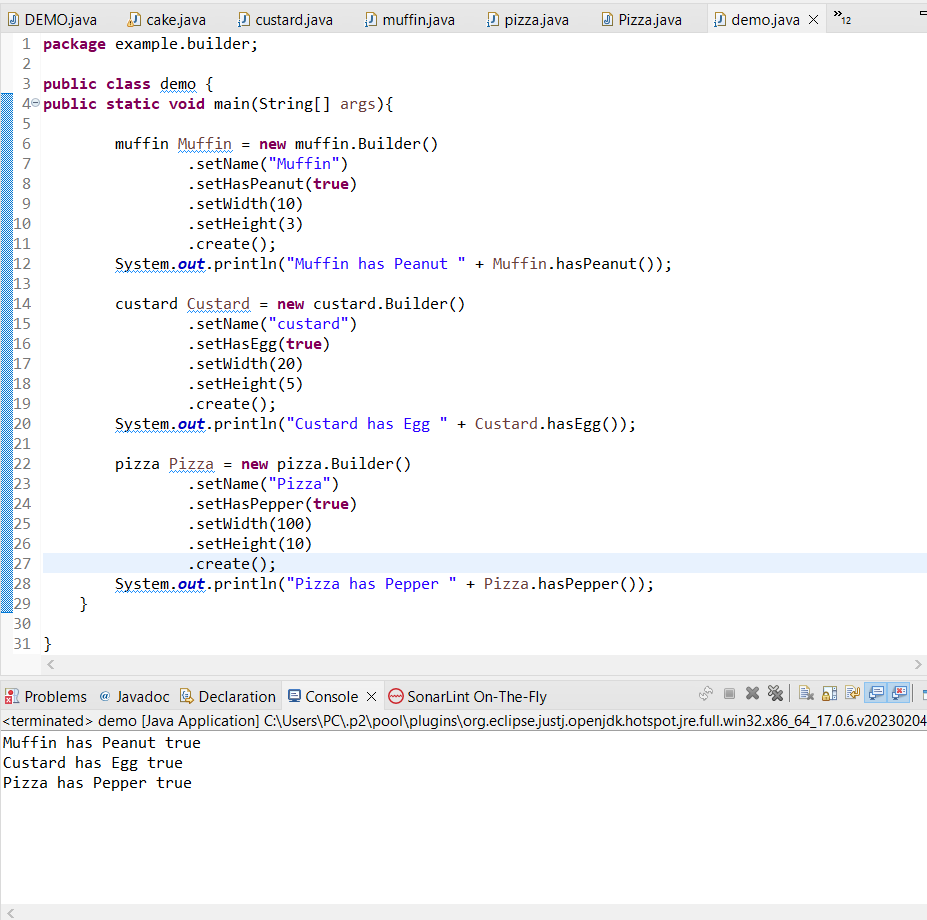
Tiếp theo, tạo một đối tượng Custard bằng cách sử dụng builder của lớp Custard và in ra thông tin về việc món custard có chứa trứng hay không.

Cuối cùng, tạo một đối tượng Pizza bằng cách sử dụng builder của lớp Pizza và in ra thông tin về việc món pizza có chứa hạt tiêu đen hay không.



Hình 2.20: Class demo

**Kết quả thực hiện**



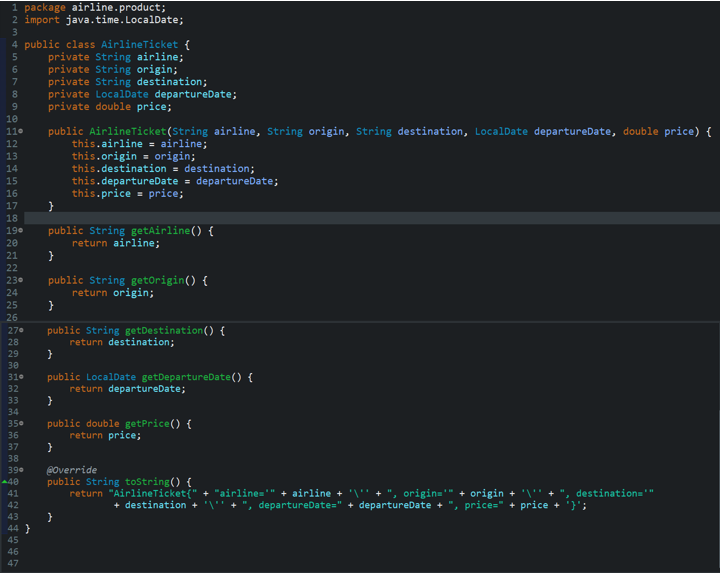
Hình 2.21: Kết quả thực hiện đã tạo Object thành công

#### **2.7.4. Trần Chí Mỹ**

**Ví dụ DEMO tạo Vé Máy Bay (AirlineTicket) sử dụng Builder Design Pattern**

* Ta sẽ xây dựng một ứng dụng đặt vé máy bay và cần tạo ra các đối tượng vé máy bay với các thuộc tính khác nhau. Trong ví dụ này, chúng ta sẽ tạo ra các đối tượng vé máy bay với các thuộc tính như hãng hàng không, điểm đi, điểm đến, ngày đi và giá vé.
* Đối tượng AirlineTicket có các thuộc tính sau: airline, origin, destination, departureDate, price. Sử dụng Builder Design Pattern để tạo một đối tượng AirlineTicket thay vì tạo bằng set thông thường.
* Ta sẽ tiến hành tạo 4 thành phần chính:

+ ***Đối tượng Product (Vé máy bay):*** Đây sẽ là đối tượng ta cần thiết kế Builder để khởi tạo nó



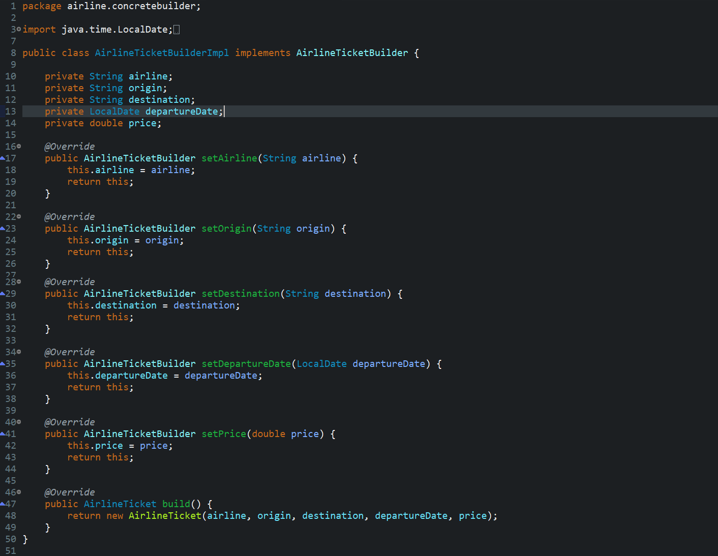
Hình 2.22: Thành phần Product (Airline Ticket)

*+* ***Đối tượng Builder(Vé máy bay Builder):*** Đây là một interface khai báo các phương thức nhận dữ liệu để tạo một product



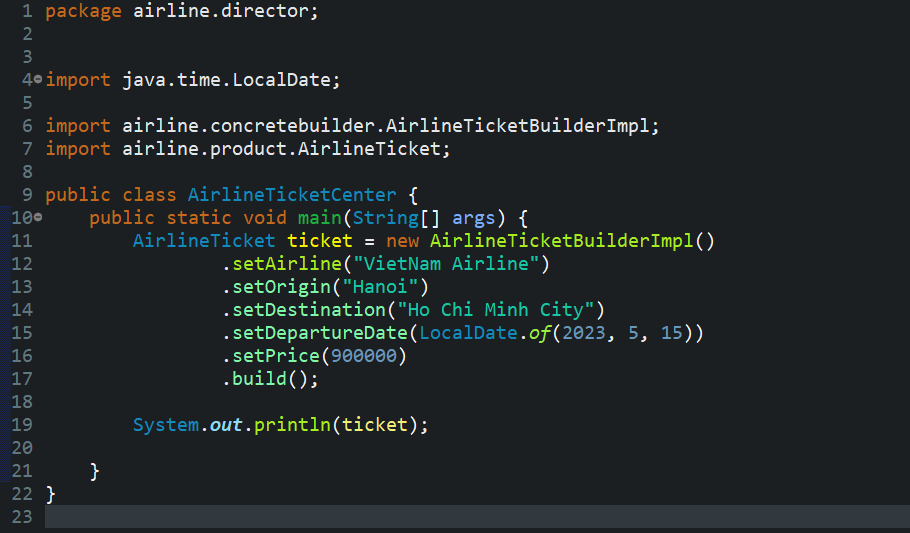
Hình 2.23: Thành phần Builder (AirlineTickerBuilder)

+ ***Đối tượng ConcreteBuilder (AirlineTicketBuilderImpl):*** Đây là một lớp cụ thể kế thừa từ AirlineTicketBuilder, và cài đặt chi tiết cách tạo ra đối tượng AirlineTicket. Nó xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời cung cấp các phương thức để trả về các thể hiện mà nó đã tạo ra trước đó.



Hình 2.24: Thành phần ConcreteBuilder (AirlineTicketBuilderImpl)

+ ***Đối tượng Director/Client (AirlineTicketCenter)***: Đây là nơi sẽ tạo ra đối tượng AirlineTicket



Hình 2.25: Thành phần Director (AirlineTicketCenter)

**Kết quả**

Ta đã tạo thành công một Object ticket bằng Builder Design Pattern

Hình 2.26: Tạo thành công một đối tượng bằng Builder Design Pattern

## **III. Façade Design Pattern (“Mặt tiền”)**

1. **Giới thiệu về Façade Design Pattern**

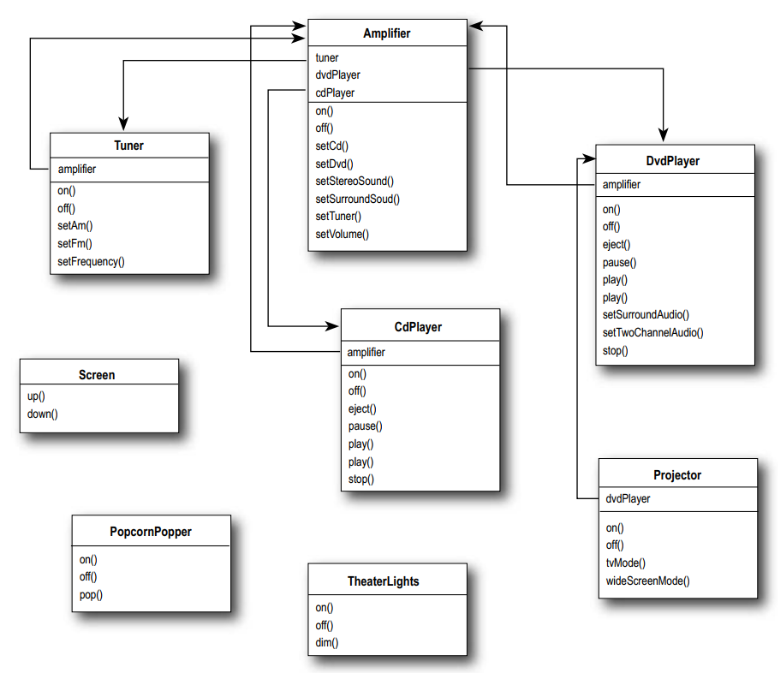
* Façade Design Pattern là một mẫu thiết kế thuộc loại Structural Design Pattern. Nó cung cấp một giao diện đơn giản để truy cập vào hệ thống phức tạp hơn bằng cách giảm thiểu sự phức tạp của hệ thống và cung cấp một giao diện đơn giản hơn cho người dùng.
* Façade Design Pattern được sử dụng để che giấu sự phức tạp của một hệ thống và đóng gói tất cả các phần phức tạp của hệ thống đó vào một interface đơn giản hơn. Điều này giúp cho người sử dụng có thể dễ dàng tương tác với hệ thống mà không cần biết chi tiết về cách hoạt động bên trong.

1. **Đặt vấn đề**

* Thông thường khi không sử dụng Façade Design Pattern thì khi Client muốn thưc hiện một hành động mà cần sự kết hợp giữa các đối tượng khác nhau thì Client phải tạo ra thực hiện việc tao ra các đối tương từ class đó sau đó gọi ra các hàm để thực hiện các bước để hoàn thành các hành động (Vì trong OOP các class thường tuân thủ theo tính chất “Single Reposibility” nghĩa là một class sẽ có một trách nhiệm).

**Ví dụ:** Việc xây dựng một rạp phim gia đình.

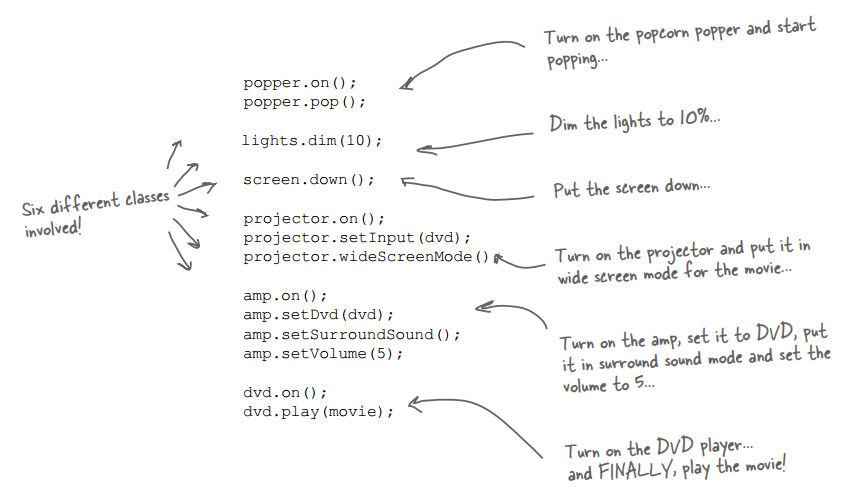
* Một rạp phim có một hệ thống với đầu DVD, hệ thống video chiếu, màn hình tự động, âm thành vồm. Đây là những class cần trong hệ thống này, mỗi class có các hàm tuân thủ theo quy tắc “Single Reposibility”, nghĩa là một class, function đảm nhiều một chức năng.



Hình 3.1: Class Diagram

Quy trình thực hiện xem một bộ phim sẽ như sau:

1. Bật chế độ ăn bỏng ngô
2. Giảm ánh sáng
3. Đặt màn hình xuống
4. Bật máy chiếu
5. Bật bộ khuếch đại, cài đặt DVD, điều chỉnh âm thanh
6. Bật đầu DVD
7. Bắt đầu phát DVD player

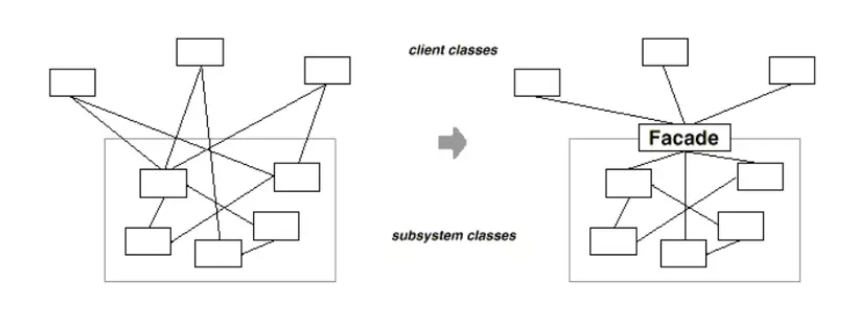


Việc Client sẽ gọi lai tất cả hàm như trên phải khởi tạo các đối tượng từ class chứa nó điều này sẽ có những nhược điểm sau:

+ Tăng độ phức tạp: Khi Client gọi nhiều phương thức từ các đối tượng khác nhau, code có thể trở nên rắc rối và khó hiểu.

+ Tăng sự phụ thuộc của class trong lớp Client.

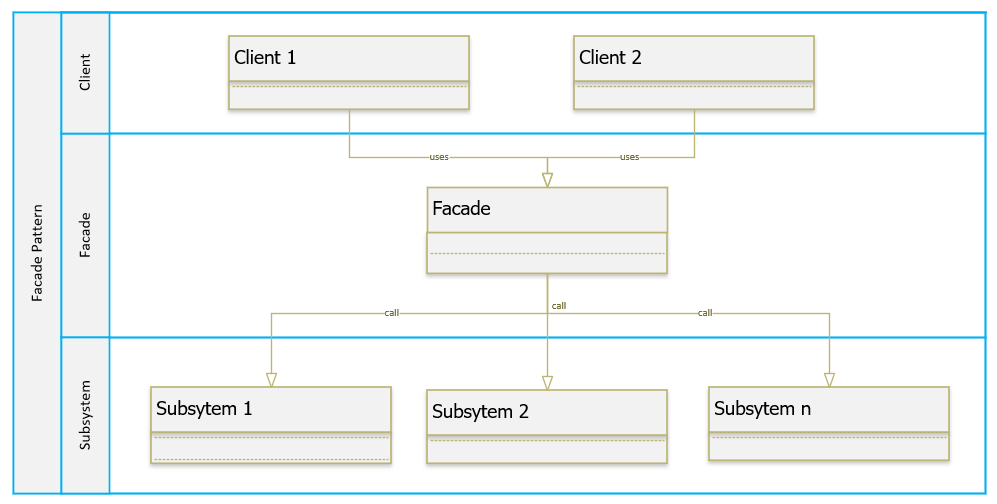
+ Khó mở rộng: Khi cần thêm các đối tượng mới hoặc thay đổi, Client phải thay đổi code và các đối tượng liên quan.



Hình 3.2: So sánh 2 cách triển khai

Cách tốt hơn để triển khai hệ thống trên đó là sử dụng Façade Design Pattern.

1. **Sơ đồ Diagram của Façade Design Pattern**



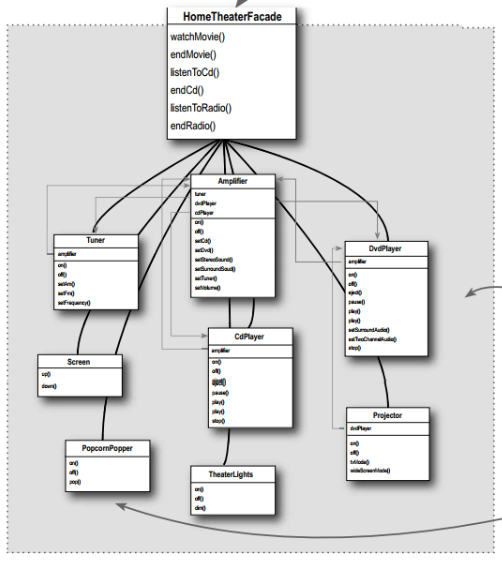
Hình 3.3: Sơ đồ Diagram của Façade Design Pattern

Sơ đồ diagram của Façade Design Pattern bao gồm các thành phần sau:

* Façade: Là đối tượng nằm giữa Client với các đối tượng khác trong hệ thống. Nó cung cấp các method và các method này gọi đến các hàm của các đối tượng khác trên hệ thống để thực hiện công việc mà Client cần.
* Subsystems: Là các đối tượng có liên quan trong hệ thống. Chúng có thể là các lớp, các phương thức.
* Client: Là đối tượng sử dụng Façade để tương tác với hệ thống. Nó không cần biết chi tiết các đối tượng trông hệ thống, chỉ cần gửi yêu cầu đến Façade để thực hiện các chức năng cần thiết.

1. **Triển khai Façade Design Pattern**

* Sử dụng Façade Pattern sẽ giúp thiểu sự phức tạp và phụ thuộc giữa Client và các đối tượng. Lúc này ta sẽ tạo ra một class Façade và client sẽ giao tiếp qua lớp này để thực hiện các hành động mà không cần biết phía sau nó sẽ triển khai như thế nào.
* Theo ví dụ trên, lúc này để tạo một Façade cho hệ thống rap hát chúng ta sẽ tạo một lớp HomeThreaterFacade mới hiển thị một vài phương thức đơn giản như watchMovie().



* Client lúc này muốn thực hiện xem phim chỉ cần gọi phương thức watchMovie bên trong class HomeThreadFacade, lúc này Façade sẽ triển khai các hành động: giao tiếp với đèn, cài đặt DVD, máy chiếu, bộ khuếch đại,…Client lúc này chỉ làm việc trực tiếp với class Façade và chỉ new đối tượng từ class này, nó còn giúp client giảm sự phụ thuộc giữa các class vì không cần phải new các đối tượng của các class để thực hiện hàng loạt các hàm như trước.
* Với Façade Pattern ta đã biến một hệ thống con phức tạp và làm cho nó dễ sử dụng hơn bằng cách triển khai lớp Façade cung cấp một giao diện hợp lý.

1. **Ưu điểm và nhược điểm của Façade Design Pattern**

* **Nhược điểm:**

+ Không phù hợp với hệ thống đơn giản: Façade thường sử dụng trong các hệ thống phức tạp, vì trong các hệ thống đơn giản, sử dụng Façade có thể làm tăng sự phức tạp và chi phí của hệ thống.

+ Không phù hợp cho các ứng dụng đòi hỏi tốc độ xử lý cao: Việc sử dụng Facade có thể làm giảm hiệu suất của hệ thống đối với các ứng dụng đòi hỏi tốc độ xử lý cao, vì nó tạo ra một lớp trung gian giữa Client và các đối tượng khác trong hệ thống.

* **Ưu điểm:**

+ Giảm thiểu sự phức tạp: Façade cung cấp một giao diện đơn giản hơn cho Client để tương tác với các đối tượng khác trong hệ thống, giúp giảm thiểu sự phức tạp của hệ thống và làm cho mã nguồn dễ dàng quản lý, bảo trì hơn.

+ Tăng tính đơn giản: Khi sử dụng Façade, Client không cần phải biết chi tiết về các đối tượng khác trong hệ thống, chỉ cần gửi yêu cầu đến Façade để thực hiện chức năng cần thiết.

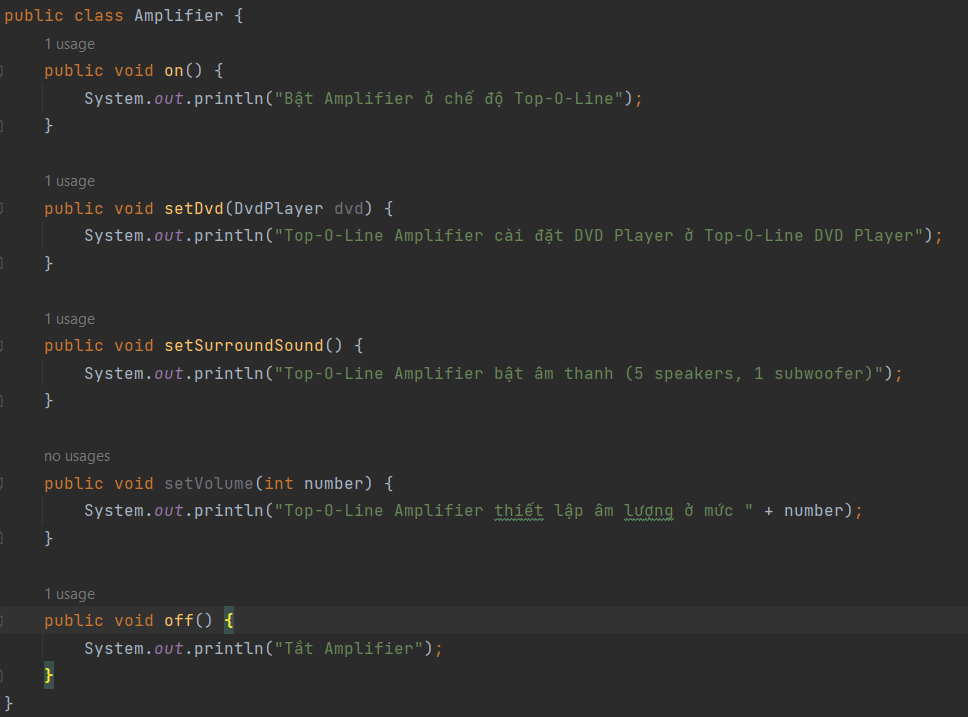
+ Tăng tính linh hoạt: Façade cho phép thay đổi các phương thức hoặc các đối tượng khác trong hệ thống mà không cần ảnh hướng đến Client, giúp tăng tính linh hoạt trong việc phát triển và bảo trì hệ thống.

+ Với mỗi hệ thống con chúng ta có thể triển khai nhiều hơn một Façade điều này giúp tăng tính đơn giản cho Client. Nó cung giúp tách riêng các chức năng khác nhau trong hệ thống và làm cho mã nguồn dễ quản lý và bảo trì hơn

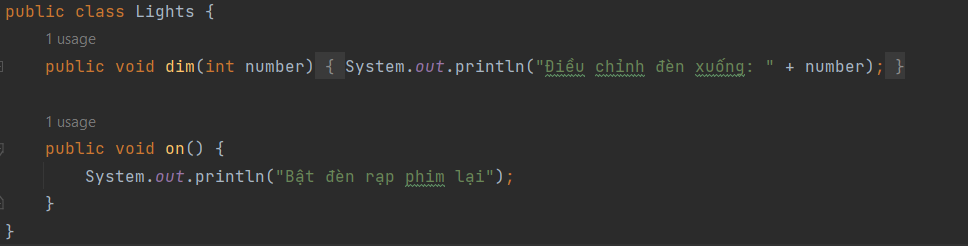
1. **Demo**

Em sẽ thực hiện trên ví dụ xây dựng một rạp phim gia đình bằng Facade Pattern

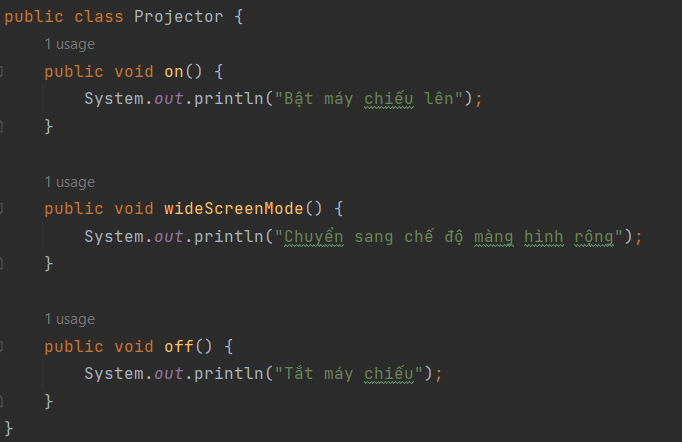
* Subsystems bao gồm các Model của hệ thống rạp phim bao gồm: Screen, Projector, Lights, DVD Player, PopCorn Popper, Amplifier trong các class này sẽ có các method thực hiện các chức năng trong rạp phim.



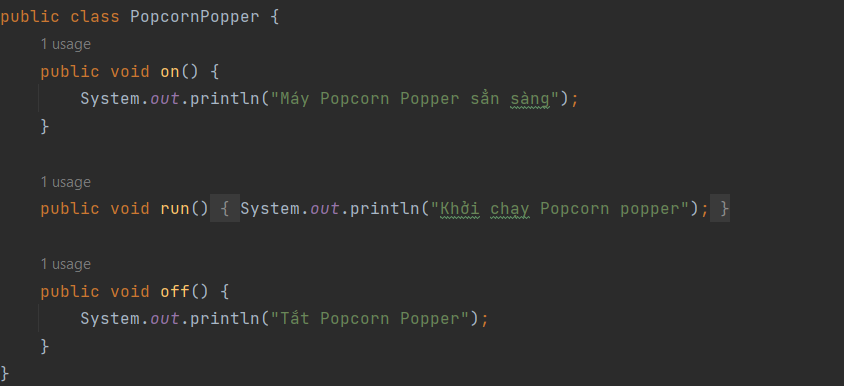
Hình 3.4: Class Amplifier



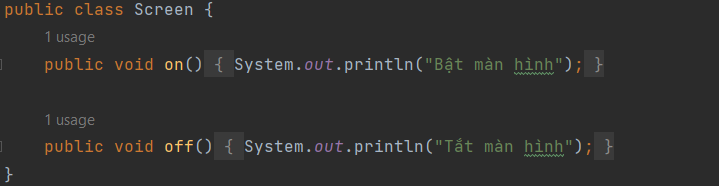
Hình 3.5: Class Lights



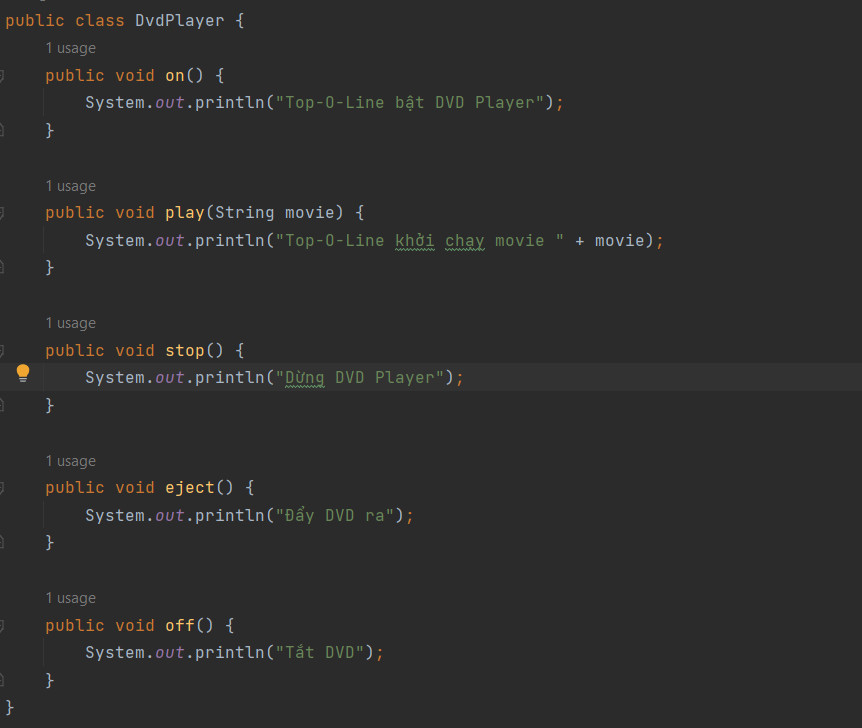
Hình 3.6: Class Projector



Hình 3.7: Class PopcornPopper

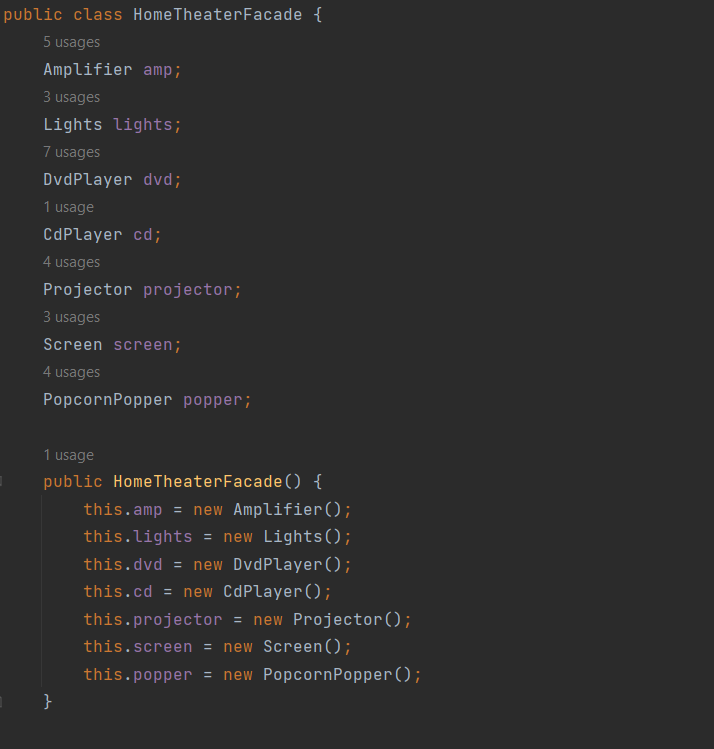


Hình 3.8: Class Screen



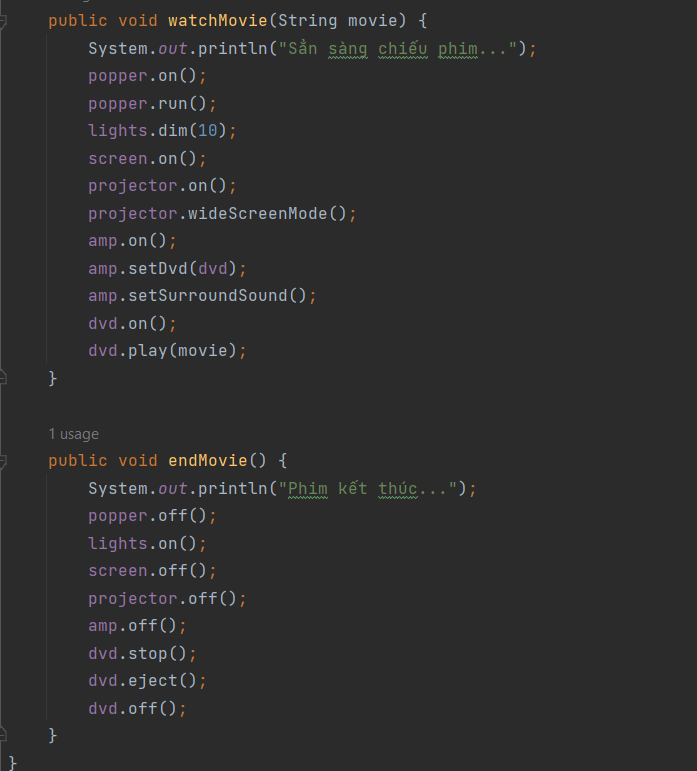
Hình 3.9: Class DvdPlayer

* Façade: Trong class này, đầu tiên em sẽ khai báo và khởi tạo các Subsystems ở phía trên để có quyền truy cập vào tất cả các thành phần trong các class này.



Hình 3.10: Class Facade trong hệ thống rap phim

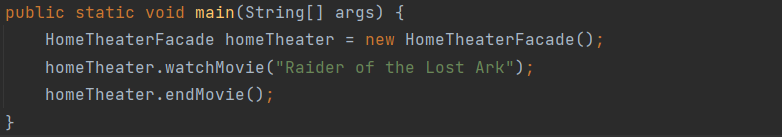
Sau đó trong class này ta có các phương thức **watchMovie()** và **endMovie()**, trong các phương thức này em gọi lại các phương thức trong các class Subsystems ở trên để thực hiện chức năng xem phim và tắt phim.



Hình 3.11: Thêm các phương thức watchMovie và endMovie

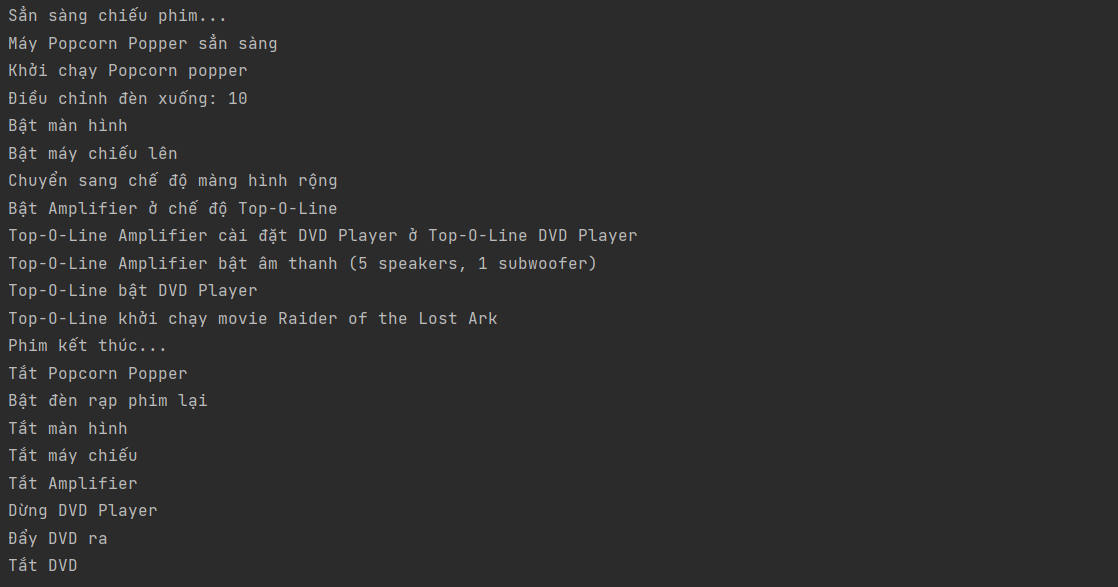
**Client:**

Trong hàm main lúc này chỉ cần khởi tạo class Facade và gọi hai phương thức watchMovie() và endMovie() là có thể thực hiện việc xem phim và tắt phim



Hình 3.12: Hàm main khởi tạo homeTheater

**Kết quả:**



Hình 3.13: Kết quả đạt được

# **TÀI LIỆU THAM KHẢO**

[1] <https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/>

[2] <https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/>

[3]<https://viblo.asia/p/minh-da-xai-design-pattern-cho-muc-dich-ca-nhan-nhu-the-nao-phan-1singleton-va-builder-WAyK8GV95xX>

[4] <https://toihocdesignpattern.com/chuong-7-adapter-va-facade-pattern-tro-nen-thich-nghi-phan-2.html>