**ĐẠI HỌC ĐÀ NẴNG**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM**

**KHOA TIN HỌC**

A picture containing text, alcohol

Description automatically generated****

**BÁO CÁO BÀI TẬP JAVASCRIPT**

**Tên đề tài:**

**BÀI TẬP NGHIÊN CỨU CHI TIẾT VỀ**

**DESIGN PATTERNS**

**GV Hướng Dẫn: Th.S Mai Hà Thi**

**Sinh Viên Thực Hiện:**

Nguyễn Hữu Tín

*Đà Nẵng, tháng 4 năm 2025*

# MỤC LỤC

[CHƯƠNG 1: TỔNG QUAN VỀ DESIGN PATTERNS 5](#_Toc196747839)

[1.1. Khái niệm: 5](#_Toc196747840)

[1.2. Design Patterns hỗ trợ developers như thế nào: 5](#_Toc196747841)

[1.2.1. Tăng tốc độ phát triển phần mềm: 5](#_Toc196747842)

[1.2.2. Code tường minh, dễ dàng team work: 5](#_Toc196747843)

[1.2.3. Tái sử dụng code: 5](#_Toc196747844)

[1.2.4. Hạn chế lỗi tiềm ẩn, dễ dàng nâng cấp: 6](#_Toc196747845)

[1.3. Phân loại Design Patterns: 6](#_Toc196747846)

[1.3.1. Creational Pattern (Nhóm khởi tạo): 6](#_Toc196747847)

[1.3.2. Structural Pattern (Nhóm cấu trúc): 7](#_Toc196747848)

[1.3.3. Behavioral Pattern (Nhóm hành vi): 8](#_Toc196747849)

[CHƯƠNG 2: TÌM HIỂU CHI TIẾT 5 LOẠI DESIGN PATTERNS CÓ THỂ ỨNG DỤNG TRONG JAVASCRIPT 10](#_Toc196747850)

[2.1. Singleton Pattern: 10](#_Toc196747851)

[2.1.1. Giới thiệu: 10](#_Toc196747852)

[2.1.2. Kiến trúc: 10](#_Toc196747853)

[2.1.3. Kịch bản giả định và mã nguồn giả lập: 11](#_Toc196747854)

[2.2. Observer Pattern: 11](#_Toc196747855)

[2.2.1. Giới thiệu: 11](#_Toc196747856)

[2.2.2. Kiến trúc 12](#_Toc196747857)

[2.2.3. Kịch bản giả định và mã nguồn giả lập: 13](#_Toc196747858)

[2.3. Decorator Pattern: 15](#_Toc196747859)

[2.3.1. Giới thiệu: 15](#_Toc196747860)

[2.3.2. Kiến trúc: 15](#_Toc196747861)

[2.3.3. Kịch bản giả định và mã nguồn giả lập: 15](#_Toc196747862)

[2.4. Factory Method Pattern: 17](#_Toc196747863)

[2.4.1. Giới thiệu: 17](#_Toc196747864)

[2.4.2. Kiến trúc: 18](#_Toc196747865)

[2.4.3. Kịch bản giả định và mã nguồn giả lập: 18](#_Toc196747866)

[2.5. Singleton Pattern: 20](#_Toc196747867)

[2.5.1. Giới thiệu: 20](#_Toc196747868)

[2.5.2. Kiến trúc: 20](#_Toc196747869)

[2.5.3. Kịch bản giả định và mã nguồn giả lập: 21](#_Toc196747870)

[CHƯƠNG 3: TỔNG KẾT 24](#_Toc196747871)

[3.1. Singleton Pattern: 24](#_Toc196747872)

[3.1.1. Mục đích: 24](#_Toc196747873)

[3.1.2. Ưu & Nhược điểm: 24](#_Toc196747874)

[3.2. Observer Pattern: 24](#_Toc196747875)

[3.2.1. Mục đích: 24](#_Toc196747876)

[3.2.2. Ưu & Nhược điểm: 24](#_Toc196747877)

[3.3. Decorator Pattern: 25](#_Toc196747878)

[3.3.1. Mục đích: 25](#_Toc196747879)

[3.3.2. Ưu & Nhược điểm: 25](#_Toc196747880)

[3.4. Factory Method Pattern: 25](#_Toc196747881)

[3.4.1. Mục đích: 25](#_Toc196747882)

[3.4.2. Ưu & Nhược điểm: 26](#_Toc196747883)

[3.5. Facade Pattern: 26](#_Toc196747884)

[3.5.1. Mục đích: 26](#_Toc196747885)

[3.5.2. Ưu & Nhược điểm: 27](#_Toc196747886)

# DANH MỤC ẢNH

[CHƯƠNG 1: TỔNG QUAN VỀ DESIGN PATTERNS 5](#_Toc196748089)

[Hình 1.1. Nhóm Creational Pattern 6](#_Toc196748090)

[Hình 1.2. Nhóm Structural Pattern 7](#_Toc196748091)

[Hình 1.3. Nhóm Behavioral Pattern 8](#_Toc196748092)

[CHƯƠNG 2: TÌM HIỂU CHI TIẾT 5 LOẠI DESIGN PATTERNS CÓ THỂ ỨNG DỤNG TRONG JAVASCRIPT 10](#_Toc196748093)

[Hình 2.1. Kiến trúc Singleton Pattern 10](#_Toc196748094)

[Hình 2.2. Mã nguồn Singleton Pattern 11](#_Toc196748095)

[Hình 2.3. Kiến trúc Observer Pattern 12](#_Toc196748096)

[Hình 2.4. Mã nguồn Observer Pattern 15](#_Toc196748097)

[Hình 2.5. Mã nguồn Decorator Pattern 17](#_Toc196748098)

[Hình 2.6. Kiến trúc Factory Method Pattern 18](#_Toc196748099)

[Hình 2.7. Mã nguồn Factory Method Pattern 19](#_Toc196748100)

[Hình 2.8. Kiến trúc Facade Pattern 20](#_Toc196748101)

[Hình 2.9. Mã nguồn Singleton Pattern 23](#_Toc196748102)

# CHƯƠNG 1: TỔNG QUAN VỀ DESIGN PATTERNS

## 1.1. Khái niệm:

Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày.

Design patterns là một kỹ thuật trong lập trình hướng đối tượng, không phải là ngôn ngữ cụ thể nào cả, nó là một kĩ thuật lập trình. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác. Nó khá quan trọng và mọi lập trình viên muốn giỏi đều phải biết, nó cung cấp cho ta các “mẫu thiết kế”, giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà ta gặp phải có thể ta sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu.

Design Pattern giúp ta giải quyết vấn đề một cách tối ưu nhất, cung cấp cho ta các giải pháp trong lập trình OOP.

## 1.2. Design Patterns hỗ trợ developers như thế nào:

### 1.2.1. Tăng tốc độ phát triển phần mềm:

Trong quá trình phát triển ứng dụng, việc sử dụng design patterns cho phép các developers có một công cụ để giải quyết các vấn đề thông dụng trong thiết kế phần mềm. Kể cả khi không gặp phải những vấn đề đó, việc nắm vững design patterns cũng rất hữu ích khi nó giúp developers thấy được cách giải quyết vấn đề thông qua việc ứng dụng các nguyên tắc thiết kế hướng đối tượng.

### 1.2.2. Code tường minh, dễ dàng team work:

Bên cạnh đó, design patterns định nghĩa một ngôn ngữ chung mà developers có thể dùng để giao tiếp hiệu quả hơn. Chẳng hạn, chỉ cần nêu tên một pattern, tất cả những đồng đội trong nhóm đều sẽ hình dung được cấu trúc, ý tưởng đằng sau đó và cách ứng dụng nó. Tối ưu thời gian phát triển ý tưởng hơn vì hạn chế được thời gian giải thích.

### 1.2.3. Tái sử dụng code:

Về phía dự án phần mềm, design patterns giúp developers có thể dễ dàng tái sử dụng và mở rộng code với các giải pháp tối ưu đã được kiểm chứng để giải quyết những vấn đề thông thường trong phát triển phần mềm. Do đó, khi gặp vấn đề trong xây dựng phần mềm, developers có thể coi design patterns như là kim chỉ nam để giúp mình giải quyết những vấn đề thay vì tự đi tìm kiếm giải pháp (mà có thể chưa có sự kiểm chứng kĩ càng).

### 1.2.4. Hạn chế lỗi tiềm ẩn, dễ dàng nâng cấp:

Ngoài ra, việc sử dụng lại các design patterns còn giúp developers tránh các vấn đề tiềm ẩn có thể sẽ gây ra những lỗi lớn trong tương lai, cùng với đó, điều đó cũng giúp dự án dễ nâng cấp và bảo trì trong tương lai hơn.

## 1.3. Phân loại Design Patterns:

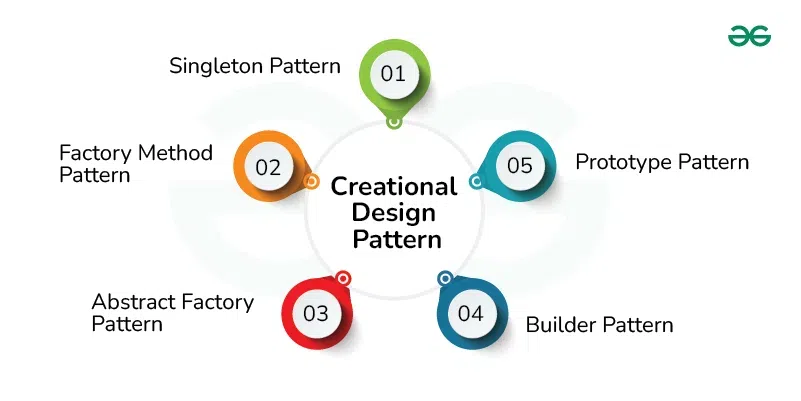
Hệ thống các mẫu design pattern hiện nay rất nhiều, nhưng thường tóm gọn bằng 23 mẫu được định nghĩa trong cuốn “Design patterns Elements of Reusable Object Oriented Software”. Hệ thống các mẫu design pattern được chia thành 3 nhóm, được phân loại theo mục đích sử dụng:

- Nhóm Creational

- Nhóm Structural

- Nhóm Behavioral

### 1.3.1. Creational Pattern (Nhóm khởi tạo):



Hình 1.1. Nhóm Creational Pattern

Nhóm Creational Pattern gồm 5 mẫu:

- [Singleton](https://viblo.asia/p/signleton-desgin-pattern-tro-thu-dac-luc-cua-developers-Qbq5QBkJKD8)

- [Factory Method](https://viblo.asia/p/factory-method-design-pattern-tro-thu-dac-luc-cua-developers-924lJBLYlPM)

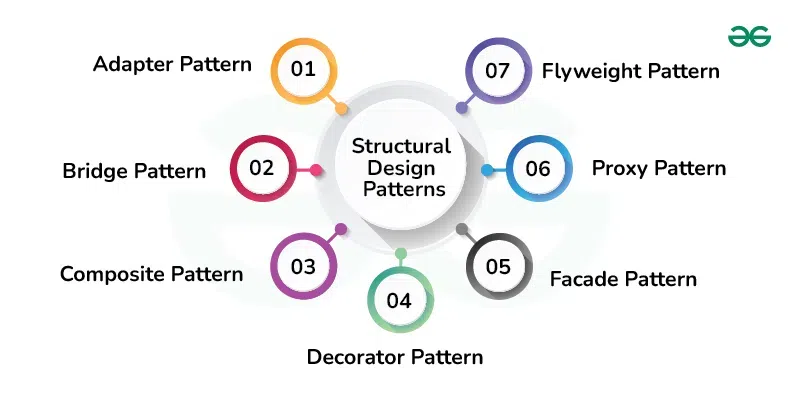
- [Abstract Factory](https://viblo.asia/p/abstract-factory-design-pattern-tro-thu-dac-luc-cua-developers-maGK7B4M5j2)

- [Builder](https://viblo.asia/p/builder-design-pattern-tro-thu-dac-luc-cua-developers-bWrZnowwlxw)

- [Prototype](https://viblo.asia/p/prototype-design-pattern-tro-thu-dac-luc-cua-developers-GrLZDBQO5k0)

Các patterns loại này cung cấp giải pháp để tạo ra các đối tượng và che giấu được logic của việc tạo ra nó thay vì tạo ra đối tượng theo cách trực tiếp (sử dụng từ khoá new). Điều này giúp chương trình trở nên mềm dẻo hơn trong việc quyết định đối tượng nào cần được tạo ra trong những tình huống khác nhau.

### 1.3.2. Structural Pattern (Nhóm cấu trúc):



Hình 1.2. Nhóm Structural Pattern

Nhóm Structural Pattern gồm 7 mẫu:

- [Adapter](https://viblo.asia/p/adapter-design-pattern-tro-thu-dac-luc-cua-developers-Az45bqYQlxY)

- [Bridge](https://viblo.asia/p/bridge-design-pattern-tro-thu-dac-luc-cua-developers-gDVK2oG2ZLj)

- [Composite](https://viblo.asia/p/composite-design-pattern-tro-thu-dac-luc-cua-developers-Qbq5QBk3KD8)

- [Decorator](https://viblo.asia/p/decorator-design-pattern-tro-thu-dac-luc-cua-developers-1VgZvQ1OKAw)

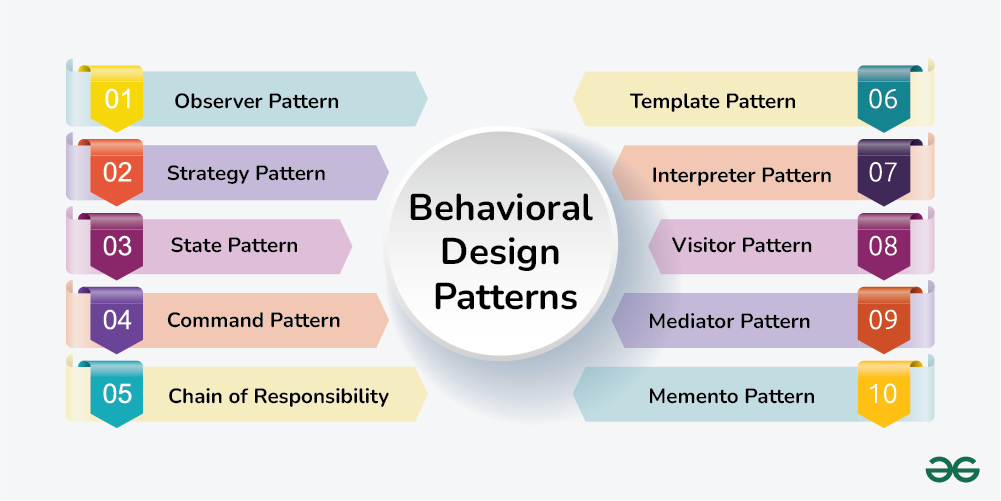
- [Facade](https://viblo.asia/p/facade-design-pattern-tro-thu-dac-luc-cua-developers-924lJBLNlPM)

- [Flyweight](https://viblo.asia/p/flyweight-design-pattern-tro-thu-dac-luc-cua-developers-maGK7B4b5j2)

- [Proxy](https://viblo.asia/p/proxy-design-pattern-tro-thu-dac-luc-cua-developers-RQqKLB2bl7z)

Những patterns loại này liên quan tới class và các thành phần của đối tượng. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng. Hệ thống càng lớn thì mẫu này càng đóng vai trò quan trọng. Ta có thể dựa vào class diagram để theo dõi mẫu này.

### 1.3.3. Behavioral Pattern (Nhóm hành vi):



Hình 1.3. Nhóm Behavioral Pattern

Nhóm Behavioral Pattern gồm 11 mẫu:

- [Iterpreter](https://viblo.asia/p/interpreter-design-pattern-tro-thu-dac-luc-cua-developers-djeZ1d43KWz)

- [Template Method](https://viblo.asia/p/template-method-design-pattern-tro-thu-dac-luc-cua-developers-Az45bqYLlxY)

- [Chain of Responsibility](https://viblo.asia/p/chain-of-responsibility-design-pattern-tro-thu-dac-luc-cua-developers-yMnKMBNDZ7P)

- [Command](https://viblo.asia/p/command-design-pattern-tro-thu-dac-luc-cua-developers-4dbZNBqkZYM)

- [Iterator](https://viblo.asia/p/iterator-design-pattern-tro-thu-dac-luc-cua-developers-jvElaNwY5kw)

- [Mediator](https://viblo.asia/p/mediator-design-pattern-tro-thu-dac-luc-cua-developers-m68Z0jVj5kG)

- [Memento](https://viblo.asia/p/memento-design-pattern-tro-thu-dac-luc-cua-developers-gGJ59BzrKX2)

- [Observer](https://viblo.asia/p/observer-design-pattern-tro-thu-dac-luc-cua-developers-gAm5y7WAZdb)

- [State](https://viblo.asia/p/state-design-pattern-tro-thu-dac-luc-cua-developers-3P0lPB9PKox)

- [Strategy](https://viblo.asia/p/strategy-design-pattern-tro-thu-dac-luc-cua-developers-bJzKmdwP59N)

- [Visitor](https://viblo.asia/p/visitor-design-pattern-tro-thu-dac-luc-cua-developers-gDVK2oGeZLj)

Nhóm này liên quan đến các quan hệ hành vi để xử lí các chức năng giữa các đối tượng trong hệ thống. Đối với các mẫu thuộc nhóm này ta có thể dựa vào collaboration và sequence diagram để theo dõi.

# CHƯƠNG 2: TÌM HIỂU CHI TIẾT 5 LOẠI DESIGN PATTERNS CÓ THỂ ỨNG DỤNG TRONG JAVASCRIPT

## 2.1. Singleton Pattern:

### 2.1.1. Giới thiệu:

Singleton là một trong số 5 design patterns thuộc nhóm Creational Design Pattern (nhóm hỗ trợ khởi tạo class). Nó đảm bảo một class chỉ có duy nhất một instance được khởi tạo và nó cung cấp phương thức truy cập đến instance đó từ mọi nơi (global access).

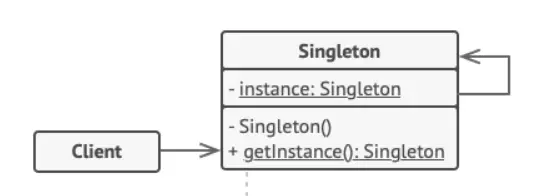
Sử dụng Singleton khi chúng ta muốn:

- Đảm bảo rằng chỉ có một instance của lớp.

- Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.

- Có thể quản lý số lượng thể hiện của một lớp trong giới hạn chỉ định.

### 2.1.2. Kiến trúc:



Hình 2.1. Kiến trúc Singleton Pattern

Cách sử dụng:

- Đặt constructor là private để client không thể khởi tạo object của class.

- Tạo một biến static private là instance của class đó để đảm bảo rằng nó là duy nhất và chỉ được tạo ra trong class đó thôi.

- Tạo một public static method trả về instance vừa khởi tạo bên trên, đây là cách duy nhất để các class khác có thể truy cập vào instance của class này.

### 2.1.3. Kịch bản giả định và mã nguồn giả lập:

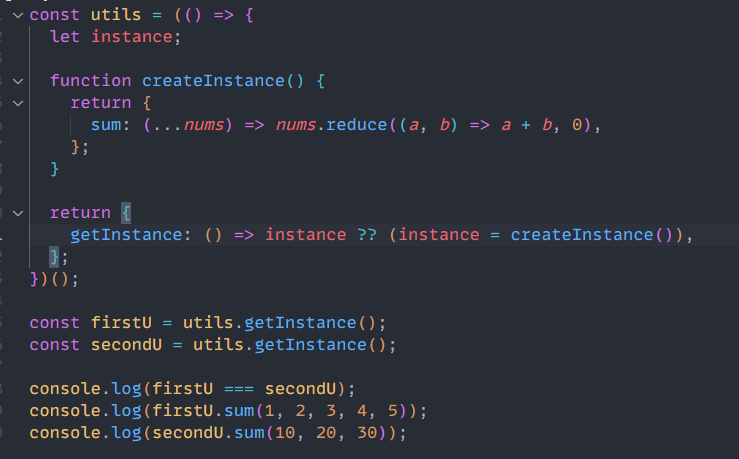
#### 2.1.3.1. Kịch bản giả định:

Tạo một bộ tiện ích (utils), có 1 hàm sum để cộng nhiều số lại với nhau.  
Yêu cầu:

- Chỉ cho phép tạo một đối tượng duy nhất để thực hiện các hàm tiện ích này.

- Các nơi khác trong chương trình gọi *getInstance()* nhiều lần cũng đều phải nhận đúng cùng một đối tượng cũ, không được tạo mới thêm.

#### 2.1.3.2. Mả nguồn giả lập:



Hình 2.2. Mã nguồn Singleton Pattern

## 2.2. Observer Pattern:

### 2.2.1. Giới thiệu:

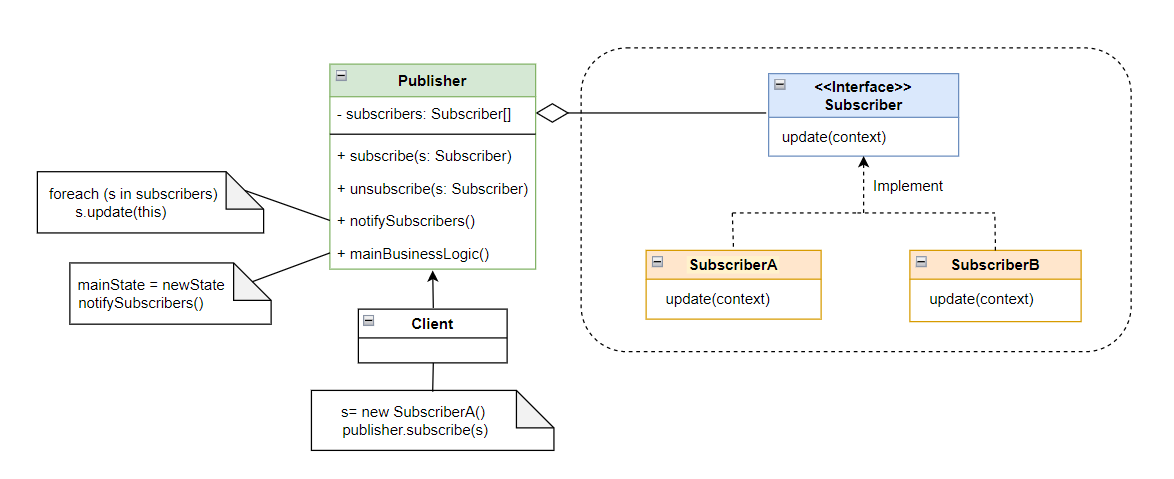
- Observer Pattern là một mẫu thiết kế thuộc nhóm Behavioral Pattern.

- Định nghĩa mối phụ thuộc một - nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất cả các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

- Chúng giống như việc khi ta đăng ký hay nhấn chuông thông báo 1 kênh Youtube, thì khi kênh đó có video mới (thay đổi trạng thái), chúng sẽ gửi thông báo (một cách tự động) đến chúng ta.

### 2.2.2. Kiến trúc



Hình 2.3. Kiến trúc Observer Pattern

Các thành phần trong mô hình:

**- Publisher (subject):** là lớp cần lắng nghe. Khi có sự kiện, Publisher sẽ thông báo cho **Subscriber (observer)**

Khi một sự kiện mới xảy ra, publisher xem qua danh sách đăng ký và gọi phương thức thông báo được khai báo trong subsrciber interface trên từng subscriber object.

**- Subscriber:** là interface để Publisher báo cáo mỗi khi có sự kiện.

Trong hầu hết các trường hợp, nó bao gồm một update method duy nhất.

**- ConcreteSubscriber (SubscriberA,B...):** cài đặt cụ thể hành động lớp cần làm khi nhận sự kiện của Publisher.

- Thông thường, subscriber cần một số thông tin theo ngữ cảnh để xử lý bản cập nhật một cách chính xác. Vì lý do này, publisher thường chuyển một số dữ liệu ngữ cảnh làm đối số của phương thức thông báo. Publisher có thể chuyển chính nó làm đối số, cho phép subscriber tìm nạp trực tiếp bất kỳ dữ liệu bắt buộc nào

**- Client** là người sử dụng Observer. Danh sách subscriber được biên dịch động: các đối tượng có thể bắt đầu hoặc dừng nghe thông báo trong thời gian chạy, tùy thuộc vào hành vi mong muốn của ứng dụng. Trong cách triển khai này, lớp Editor không tự duy trì danh sách subscription. Nó ủy thác công việc này cho đối tượng trợ giúp đặc biệt dành riêng cho việc đó. Việc thêm Subscriber mới vào chương trình không yêu cầu thay đổi đối với các class Publisher hiện có, miễn là chúng hoạt động với tất cả subscriber thông qua cùng một giao diện.

### 2.2.3. Kịch bản giả định và mã nguồn giả lập:

#### 2.2.3.1. Kịch bản giả định:

Giả sử ta đang mô phỏng một hệ thống **gửi thông báo bài viết mới** cho người đăng ký.

- Observable là **người đăng bài** (hay "Publisher").

- Observer là **người đăng ký nhận bài viết** (hay "Subscriber").

Mỗi lần Publisher có bài mới, họ sẽ **gửi thông báo đến tất cả người đăng ký**.

**Các bước chi tiết:**

1. Xây dựng đối tượng "Publisher" (Observable):

Observable có **danh sách observers** (danh sách người đã đăng ký).

Có 3 chức năng:

**- subscribe(observer)**: Thêm một người vào danh sách đăng ký.

**- unsubscribe(observer)**: Xoá người đó ra khỏi danh sách.

**- notify(message)**: Gửi thông báo cho tất cả những người đang đăng ký.

2. Xây dựng đối tượng "Subscriber" (Observer):

- Observer đại diện cho **mỗi người đăng ký**.

Mỗi Observer có 1 tên (name) và một phương thức update(message) để **xử lý khi nhận thông báo**.

3. Tạo và đăng ký các Observer:

user1, user2, user3 được tạo ra từ lớp Observer.

Các user này sẽ **gọi** *subscribe()* để nhận tin từ publisher.

4. Một người huỷ đăng ký:

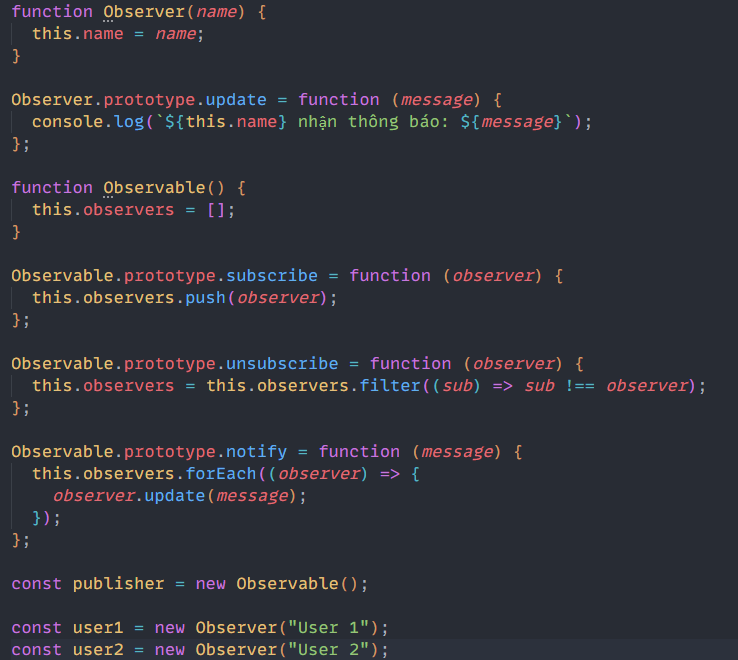
user1 **gọi** *unsubscribe()* để không còn nhận tin.

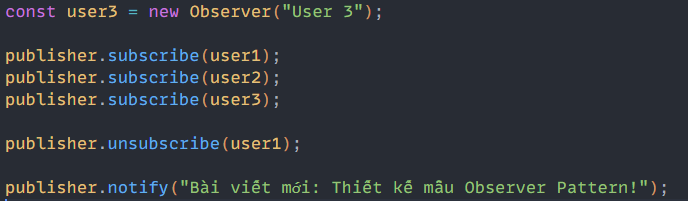
5. Gửi thông báo:

Khi publisher.notify("Bài viết mới...") được gọi:

Tất cả những user còn trong danh sách (user2 và user3) sẽ được gọi hàm update(message) và in ra thông báo.

#### 2.2.3.2. Mả nguồn giả lập:





Hình 2.4. Mã nguồn Observer Pattern

## 2.3. Decorator Pattern:

### 2.3.1. Giới thiệu:

- Phân loại: Structural pattern

- Bí danh: Wrapper

- Mục đích: Mở rộng tính năng của một object tại run-time, decorator cung cấp một sự thay thế linh hoạt cho subclass để mở rộng tính năng của object

- Tần suất sử dụng: bình thường

### 2.3.2. Kiến trúc:

Các thành phần trong mô hình:

- **Component**: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.

- **Concrete Component**: là lớp hiện thực (implements) các phương thức của Component.

**- Decorator**:là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component interface.

- **Concrete Decorator**:là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.

- **Client**:đối tượng sử dụng Component với những yêu cầu mở rộng đính kèm.

### 2.3.3. Kịch bản giả định và mã nguồn giả lập:

#### 2.3.3.1. Kịch bản giả định:

Giả sử bạn có một hệ thống bán **cà phê**, và mỗi ly cà phê cơ bản có thể được thêm **topping** như:

- Sữa

- Đường

- Caramel

Thay vì phải tạo 1000 loại "Cà phê + Sữa + Đường + Caramel" riêng biệt, ta dùng **Decorator Pattern** để **"gói thêm"** các topping vào một đối tượng cà phê cơ bản.

Nghĩa là: **topping chỉ là cái áo khoác ngoài** cho cà phê, ta "decorate" dần dần.

#### 2.3.3.2. Mã nguồn giả lập:





Hình 2.5. Mã nguồn Decorator Pattern

## 2.4. Factory Method Pattern:

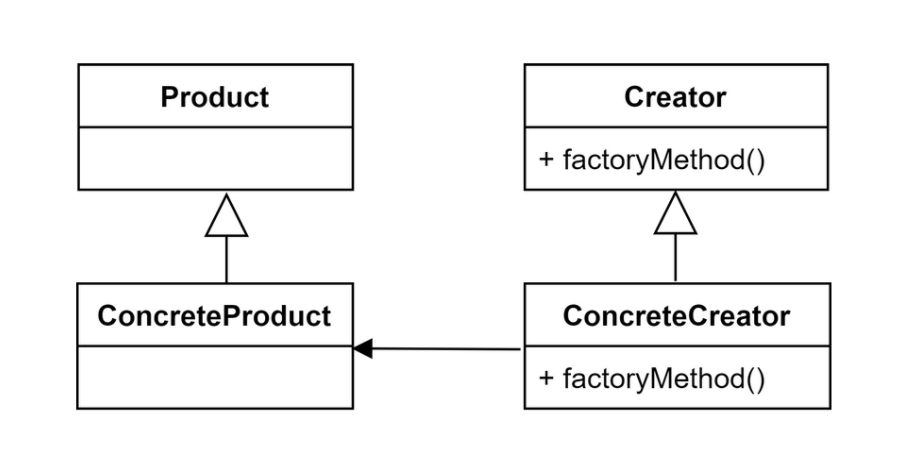
### 2.4.1. Giới thiệu:

Factory method (hay còn gọi là virtual constructor) là một mẫu thiết kế thuộc nhóm Creational Patterns – những mẫu thiết kế cho việc khởi tạo đối tượng của lớp Khi chúng ta muốn tạo ra một object của một type nào đấy, nhưng chúng ta không biết rõ mình sẽ phải tạo ra cái gì, mà nó phải dựa vào một số điều kiện business logic đầu vào để tạo ra object tương ứng, thì chúng ta có thể sử dụng Factory Method này.

**Định nghĩa:**

Factory Method cung cấp một interface, phương thức trong việc tạo nên một đối tượng (object) trong class. Nhưng để cho class con kế thừa của nó có thể ghi đè để chỉ rõ đối tượng (object) nào sẽ được tạo. Factory method giao việc khởi tao một đối tượng (object) cụ thế cho lớp con (subclass).

### 2.4.2. Kiến trúc:



Hình 2.6. Kiến trúc Factory Method Pattern

Các thành phần trong mô hình:

- **Product**: Định nghĩa một khuôn mẫu (interface) của các đối tượng mà factory method tạo ra.

- **Concreteproduct**: các lớp được cài đặt khuôn mẫu product.

- **Creator**:

+ Khai báo factory method, trả về kiểu đối tượng thuộc kiểu product. Creator cũng có thể định nghĩa một cài đặt mặc định của factory method mà giá trị trả về là một đối tượng concreteproduct mặc định.

+ Gọi factory method để tạo đổi tượng kiểu product.

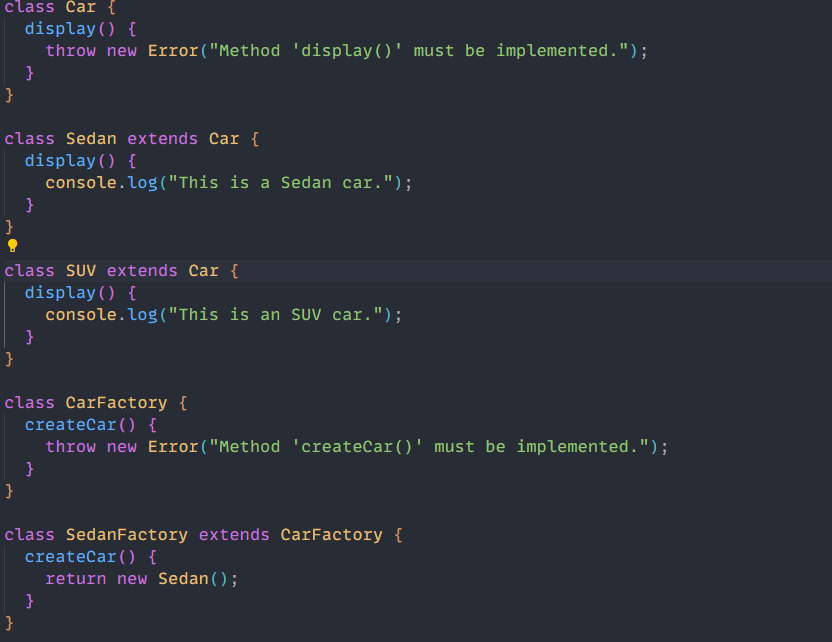
- **ConcreteCreator**: ghi đè factory method để trả về một instance của concreteproduct.

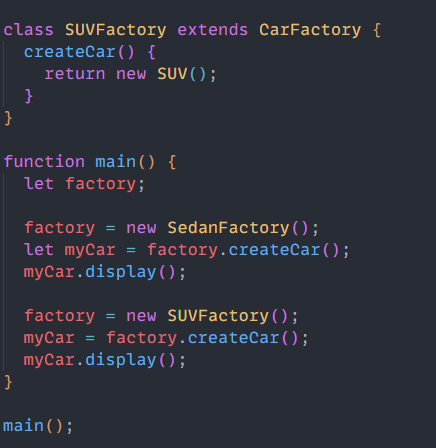
### 2.4.3. Kịch bản giả định và mã nguồn giả lập:

#### 2.4.3.1. Kịch bản giả định:

Giả sử ta đang xây dựng một ứng dụng về quản lý xe hơi và muốn tạo ra các loại xe hơi khác nhau (Ví dụ: Xe sedan, xe SUV). Thay vì trực tiếp khởi tạo các đối tượng xe trong mã nguồn, ta có thể sử dụng Factory Method để quyết định loại xe nào cần tạo.

#### 2.4.3.2. Mã nguồn giả lập:





Hình 2.7. Mã nguồn Factory Method Pattern

## 2.5. Singleton Pattern:

### 2.5.1. Giới thiệu:

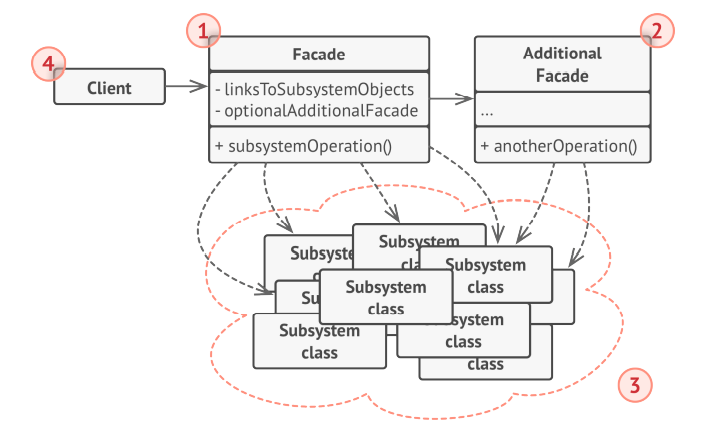
- Facade là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern).

- Facade Pattern cung cấp cho chúng ta một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

- Facade Pattern cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

- Tần suất sử dụng: khá cao

### 2.5.2. Kiến trúc:



Hình 2.8. Kiến trúc Facade Pattern

Các thành phần trong mô hình:

**- Facade**: Facade nắm rõ được hệ thống con nào đảm nhiệm việc đáp ứng yêu cầu của client, nó sẽ chuyển yêu cầu của client đến các đối tượng hệ thống con tương ứng.

**- Addition Facade**: có thể được tạo ra để tránh việc làm phức tạp một facade. Có thể được sử dụng bởi cả client và facade.

**- Complex Subsystems**: Bao gồm nhiều object khác nhau, được cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.

**- Client**: Đối tượng sử dụng Facade để tương tác với các subsystem thay vì gọi subsystem trực tiếp

Các đối tượng Facade thường là Singleton bởi vì chỉ cần duy nhất một đối tượng Facade.

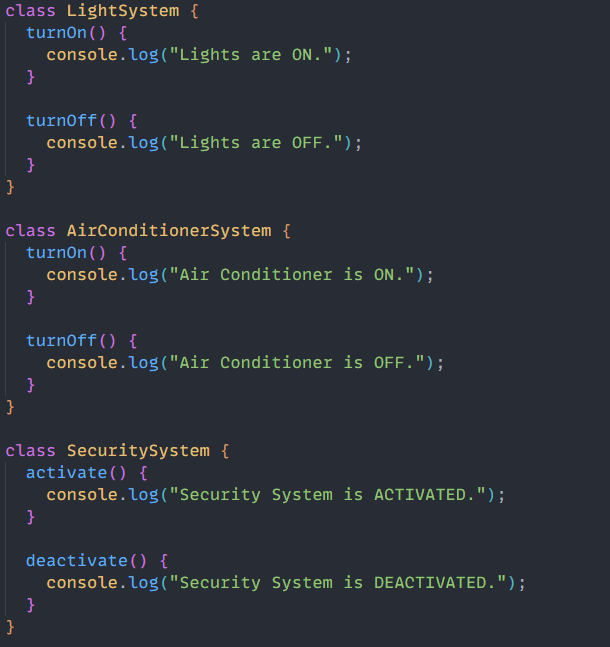
### 2.5.3. Kịch bản giả định và mã nguồn giả lập:

#### 2.5.3.1. Kịch bản giả định:

Giả sử chúng ta đang xây dựng một hệ thống **Home Automation** (Tự động hóa nhà cửa) với các thiết bị như điều hòa, ánh sáng, và hệ thống an ninh. Mỗi thiết bị có các phương thức riêng biệt để bật/tắt và kiểm tra trạng thái của nó. Nhằm cung cấp một giao diện đơn giản để người dùng có thể kích hoạt các thiết bị này cùng lúc, chẳng hạn như khi người dùng nói "Chào buổi sáng", tất cả các thiết bị sẽ bật lên.

Thay vì để người dùng phải gọi nhiều phương thức để điều khiển từng thiết bị, ta sẽ sử dụng **Facade Pattern** để tạo ra một lớp **Facade** đơn giản giúp người dùng chỉ cần gọi một phương thức duy nhất để kích hoạt tất cả các thiết bị.

#### 2.5.3.2. Mả nguồn giả lập:





Hình 2.9. Mã nguồn Singleton Pattern

# CHƯƠNG 3: TỔNG KẾT

Qua quá trình nghiên cứu chi tiết các pattern ở chương 2 ta có thể đưa ra một số kết luận về mục đích và các ưu nhược điểm của 5 loại Design Pattern ở trên như sau.

## 3.1. Singleton Pattern:

### 3.1.1. Mục đích:

Đảm bảo một class chỉ có một instance duy nhất và cung cấp phương thức toàn cục để truy cập tới object đó.

### 3.1.2. Ưu & Nhược điểm:

**Ưu điểm:**

- Có thể chắc chắn rằng một lớp chỉ có một instance.

- Có khả năng truy cập đến instance từ mọi nơi (global access).

- Đối tượng singleton chỉ được khởi tạo duy nhất một lần khi nó được yêu cầu lần đầu.

- Kiểm soát truy cập đến instance duy nhất.

- Giảm namespace.

**Nhược điểm:**

- Vi phạm Single Responsibility Principle. Mẫu này giải quyết hay vấn đề trên cùng một thời điểm.

- Singleton pattern có thể thể hiện thiết kế kém (bad design), chẳng hạn, khi các thành phần của chương trình biết quá nhiều về nhau.

- Có thể sinh ra khó khăn trong việc unit test client code của Singleton bởi nhiều test frameworks dựa vào kế thừa khi sản sinh mock objects.

## 3.2. Observer Pattern:

### 3.2.1. Mục đích:

Cho phép các đối tượng nhận thông báo khi có sự thay đổi giữa các đối tượng có mối quan hệ với nhau.

### 3.2.2. Ưu & Nhược điểm:

**Ưu điểm:**

- Đảm bảo nguyên tắc Open/Closed Principle (OCP): Cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác.

- Thiết lập mối quan hệ giữa các objects trong thời gian chạy.

- Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.

- Không giới hạn số lượng Observer

**Nhược điểm:**

- Unexpected update: Bởi vì các Observer không biết về sự hiện diện của nhau, nó có thể gây tốn nhiều chi phí của việc thay đổi Subject.

- Subscriber được thông báo theo thứ tự ngẫu nhiên.

## 3.3. Decorator Pattern:

### 3.3.1. Mục đích:

Dùng để thêm chức năng cho đối tượng mà không cần phải thay đổi cấu trúc của nó.

### 3.3.2. Ưu & Nhược điểm:

**Ưu điểm:**

- Ta có thể mở rộng hành vi của đối tượng mà không cần tạo lớp con mới.

- Ta có thể thêm hoặc xoá tính năng của một đối tượng trong lúc thực thi.

- Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.

- Single Responsibility Principle - Có thể chia nhiều cách thực thi của một phương thức trong một lớp cho nhiều lớp nhỏ hơn.

**Nhược điểm:**

- Khó để xóa một wrapper cụ thể khỏi stack.

- Khó để triển khai decorator theo cách mà phương thức của nó không phụ thuộc vào thứ tự trong stack.

## 3.4. Factory Method Pattern:

### 3.4.1. Mục đích:

**Mục đích:**

- Tạo ra một cách khởi tạo object mới thông qua một interface chung.

- Che giấu quá trình xử lý logic của phương thức khởi tạo.

- Giảm sự phụ thuộc, dễ dàng mở rộng.

- Giảm khả năng gây lỗi compile.

### 3.4.2. Ưu & Nhược điểm:

**Ưu điểm:**

- Che giấu quá trình xử lý logic của phương thức khởi tạo.

- Hạn chế sự phụ thuộc giữa creator và concrete products.

- Dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần phá vỡ các đối tượng ban đầu.

- Giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.

- Giảm khả năng gây lỗi compile, trong trường hợp chúng ta cần tạo một đối tượng mà quên khai báo lớp, chúng ta cũng có thể xử lý lỗi trong Factory và khai báo lớp cho chúng sau.

→ Vì những đặc điểm trên nên factory pattern thường được sử dụng trong các thư viện (người sử dụng đạt được mục đích tạo mới object và không cần quan tâm đến cách nó được tạo ra).

**Nhược điểm:**

- Source code có thể trở nên phức tạp hơn mức bình thường do đòi hỏi phải sử dụng nhiều class mới có thể cài đặt được pattern này.

- Việc refactoring (tái cấu trúc) một class bình thường có sẵn thành một class có Factory Method có thể dẫn đến nhiều lỗi trong hệ thống, phá vỡ sự tồn tại của clients.

- Factory method pattern lệ thuộc vào việc sử dụng private constructor nên các class không thể mở rộng và kế thừa.

## 3.5. Facade Pattern:

### 3.5.1. Mục đích:

Cung cấp một giao diện đơn giản, dễ sử dụng để ẩn đi sự phức tạp của một hệ thống hoặc một tập hợp các lớp phức tạp. Khi sử dụng **Facade Pattern**, client chỉ cần tương tác với một lớp facade duy nhất, thay vì phải tương tác với nhiều lớp khác nhau, mỗi lớp có thể có những yêu cầu phức tạp.

### 3.5.2. Ưu & Nhược điểm:

**Ưu điểm:**

- Ta có thể tách mã nguồn của mình ra khỏi sự phức tạp của hệ thống con.

- Hệ thống tích hợp thông qua Facade sẽ đơn giản hơn vì chỉ cần tương tác với Facade thay vì hàng loạt đối tượng khác.

- Tăng khả năng độc lập và khả chuyển, giảm sự phụ thuộc.

- Có thể đóng gói nhiều hàm được thiết kế không tốt bằng 1 hàm có thiết kế tốt hơn.

**Nhược điểm:**

- Class Facade của bạn có thể trở lên quá lớn, làm quá nhiều nhiệm vụ với nhiều hàm chức năng trong nó.

- Dễ bị phá vỡ các quy tắc trong SOLID (các nguyên tắc cơ bản trong OOP).

- Việc sử dụng Facade cho các hệ thống đơn giản, ko quá phức tạp trở nên dư thừa.