

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**



**MẪU THIẾT KẾ**

**BÁO CÁO**

**SE401.L11**

**Nhóm 12**

|  |  |
| --- | --- |
| **NGUYỄN DU LỊCH** | **17520096** |
| **NGUYỄN VĂN ĐÔNG** | **17520350** |
| **NGUYỄN DUY MINH** | **17520754** |

***Sinh viên thực hiện:***

***Giảng viên hướng dẫn:***

|  |  |
| --- | --- |
| **TRẦN ANH DŨNG** |  |

**LỜI CẢM ƠN**

Nhóm 12 xin chân thành gửi lời cảm ơn đến thầy Trần Anh Dũng đã gắn bó cùng nhóm 12 trong suốt thời gian học tập môn học Mẫu thiết kế. Trong quá trình học tập, Thầy đã cung cấp cho nhóm những kiến thức chuyên môn cũng như hướng dẫn nhóm thực hiện các bài Seminar để đạt được kết quả như hôm nay.

Quá trình tìm hiểu các mẫu thiết kế đã gặp không ít khó khăn, với sự giúp đỡ và hỗ trợ từ Thầy đã giúp nhóm vượt qua và hạn chế những sai sót. Tuy vậy, dù có cố gắng nhiều song công việc của Nhóm 12 vẫn có nhiều thiếu sót, kính mong Thầy xem xét và góp ý để bài báo cáo và các bài Seminar phát triển và hoàn thiện hơn.

Xin chân thành cảm ơn.

Nhóm 12 – SE401.L11

**Nguyễn Du Lịch (17520096)**

**Nguyễn Văn Đông (17520350)**

**Nguyễn Duy Minh (17520754)**

Mục lục

[I. Giới thiệu về Mẫu thiết kế (Design Pattern): 5](#_Toc60991511)

[1. Mẫu thiết kế (Design Pattern) là gì? 5](#_Toc60991512)

[2. Vai trò của Design pattern trong phát triển phần mềm: 5](#_Toc60991513)

[3. Lich sử của mẫu thiết kế: 6](#_Toc60991514)

[II. Giới thiệu các mẫu thiết kế Gang of Four 4.0 (GoF): 6](#_Toc60991515)

[1. Tổng quan: 6](#_Toc60991516)

[2. Phân loại Design Pattern theo GoF: 6](#_Toc60991517)

[III. Các mẫu thiết kế GoF 7](#_Toc60991518)

[1. Abstract Factory 7](#_Toc60991519)

[2. Builder 9](#_Toc60991520)

[3. Factory Method 11](#_Toc60991521)

[4. Prototype 13](#_Toc60991522)

[5. Singleton 15](#_Toc60991523)

[6. Adapter 16](#_Toc60991524)

[7. Bridge 18](#_Toc60991525)

[8. Composite 20](#_Toc60991526)

[9. Decorator 22](#_Toc60991527)

[10. Facade 24](#_Toc60991528)

[11. Flyweigth 25](#_Toc60991529)

[12. Proxy 27](#_Toc60991530)

[13. Chain or Responsibility 29](#_Toc60991531)

[14. Command 32](#_Toc60991532)

[15. Interpreter 35](#_Toc60991533)

[16. Iterator 37](#_Toc60991534)

[17. Mediator 39](#_Toc60991535)

[18. Memento 41](#_Toc60991536)

[19. Observer 44](#_Toc60991537)

[20. State 46](#_Toc60991538)

[21. Strategy 48](#_Toc60991539)

[22. Template Method 49](#_Toc60991540)

[23. Visitor 51](#_Toc60991541)

[IV. Tài liệu tham khảo 53](#_Toc60991542)

**NỘI DUNG CHÍNH**

# Giới thiệu về Mẫu thiết kế (Design Pattern):

## Mẫu thiết kế (Design Pattern) là gì?

- Mẫu thiết kế (design pattern) là các giải pháp tổng thể đã được nghiên cứu và 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. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết được các vấn đề trong các tình huống cụ thể.

- Một mẫu thiết kế là một cặp (vấn đề, lời giải) có thể áp dụng trong nhiều tình huống, ngữ cảnh khác nhau. Mỗi mẫu thường bao gồm các bộ phận sau:

• Tên.

• Nội dung vấn đề.

• Lời giải (phải đủ tổng quát để có thể dùng trong nhiều tình huống).

• Các hệ quả mang lại và ví dụ áp dụng.

- Lập trình viên có thể áp dụng các mẫu thiết kế này để giải quyết các vấn đề theo các khuôn mẫu nhất định hoặc các vấn đề tương tự. Các vấn đề hầu hết lập trình viêcn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu.

- Các mẫu thiết kế có thể được hiện thực ở phần lớn các ngôn ngữ lập trình. Nó giúp lập trình viên giải quyết vấn đề lập trình một cách tối ưu nhất, cung cấp cho lập trình viên các giải pháp trong lập trình hướng đối tượng (OOP).

## Vai trò của Design pattern trong phát triển phần mềm:

- Việc sử dụng các mẫu thiết kế sẽ giúp cho sản phẩm có thể hoạt động linh hoạt, lập trình viên dễ thay đổi cũng như thực hiện bảo trì sản phẩm hơn.

- Các mẫu thiết kế có khả năng cung cấp các giải pháp đã được tối ưu hóa cũng như kiểm chứng để có thể giải quyết được các vấn đề xảy ra trong phát triển phần mềm. Những giải pháp tổng quát này sẽ giúp cho lập trình viên gia tăng được cho tốc độ phát triển của phần mềm nhờ cách đưa ra được các mô hình test cũng như các mô hình phát triển đều đã qua kiểm nghiệm.

- Hạn chế các vấn đề, khó khăn không đáng có. Các mẫu thiết kế giúp giải quyết triệt để các vấn đề có sẵn và đỡ tốn kém về mặt thời gian.

- Cải thiện performance sản phẩm.

- Nhờ các khuôn mẫu có sẵn, các mẫu thiết kế giúp mọi hoạt động của các lập trình viên trong một team đều có thể chia sẻ với nhau một cách dễ dàng hơn.

## Lich sử của mẫu thiết kế:

- Khái niệm về các mẫu (pattern) lần đầu tiên được mô tả bởi Christopher Alexander trong cuốn: **“A Pattern Language: Towns, Buildings, Construction”**. Cuốn sách này mô tả một “ngôn ngữ” để có thể thiết kế nên một đô thị. Trong đó các đơn vị của “ngôn ngữ” này là các pattern.

- Năm 1987, các nền tảng của mẫu thiết kế được hai lập trình viên là Kent Beck và Ward Cunningham phát triển và bắt đầu được sử dụng trong lập trình.

- Năm 1994, sau khi cuốn sách **“Design Patterns: Elements of Reusable Object-Oriented Software”** của bộ từ Gang of Four tình bày về cách sử dụng các pattern trong lập trình được xuất bản thì mẫu thiết kế mới bắt đầu trở nên phổ biến. Cuốn sách viết về 23 mẫu thiết kế này đã giải quyết rất nhiều các vấn đề trong lập trình và nhanh chóng trở thành best-seller. Vì tên của quyển sách này quá dài mà người ta còn gọi nó là **“the book by the gang of four”** hay **“the GoF book”**. Khoảng ½ của bộ mẫu thiết kế này có nguồn gốc từ luận án tiến sĩ của Erich Gamma.

- Sau này, hàng chục mẫu thiết kế hướng đối tượng khác đã được phát minh. Phương pháp tiếp cận theo mẫu thiết kế đã trở nên rất phổ biến trong các lĩnh vực lập trình khác, vì vậy rất nhiều mẫu thiết kế khác hiện cũng tồn tại ngoài các mẫu thiết kế hướng đối tượng.

# Giới thiệu các mẫu thiết kế Gang of Four 4.0 (GoF):

## Tổng quan:

- Các mẫu thiết kế Gang of Four được biên soạn trong cuốn **“Design Patterns: Elements of Reusable Object-Oriented Software”**. Các tác giả của cuốn sách là **Erich Gamma**, **Richard Helm**, **Ralph Johnson** và **John Vlissides**, hay còn được biết đến với các tên **“Gang of Four”** hay đơn giản là ”**GoF**”. Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại. Hệ thống các mẫu design pattern được chia thành 3 nhóm: nhóm **Creational** (5 mẫu), nhóm **Structural** (7 mẫu) và nhóm **Behavioral** (11 mẫu).

## Phân loại Design Pattern theo GoF:

\*Như đã trình bày phía trên, hệ thống các mẫu design pattern GoF được chia thành 3 nhóm: nhóm **Creational** (5 mẫu), nhóm **Structural** (7 mẫu) và nhóm **Behavioral** (11 mẫu):

- Nhóm Creational (nhóm kiến tạo):

+ Abstract Factory.

+ Builder.

+ Factory Method.

+ Prototype.

+ Singleton.

- Nhóm Structural (nhóm cấu trúc):

+ Adapter.

+ Bridge.

+ Composite.

+ Decorator.

+ Façade.

+ Flyweight.

+ Proxy.

- Nhóm Behavioral (nhóm hành vi):

+ Chain of Responsibility.

+ Command.

+ Interpreter.

+ Iterator.

+ Mediator.

+ Memento.

+ Observer.

+ State.

+ Strategy.

+ Template method.

+ Visitor.

# Các mẫu thiết kế GoF

## Abstract Factory

### Tổng quan

* Tên: Abstract Factory.
* Phân loại: Abstract Factory thuộc nhóm Creational Pattern.
* Bí danh: Fabrique Abstraite.
* Mục đích: là 1 super factory, cung cấp cho các cách thức để khởi tạo các factory con, từ đó có thể khởi tạo các đối tượng liên quan đến factory con đó.

### Khả năng ứng dụng

* Ứng dụng trong 1 hệ thống có nhiều module và có ràng buộc về mặt cấu hình giữa các module này.
* Inversion of Control (IOC)
* Khi cần khởi tạo các đối tượng có hình thức khởi tạo phức tạp và không ổn định.

### Cấu trúc



Cấu tạo của Abstract Factory gồm 3 thành phần chính:

* **AbstractFactory**: lớp này có trách nhiệm khởi tạo các factory con trong hệ thống.
* **Factory con**: khởi tạo 1 loại đối tượng nhất định trong hệ thống. Ví dụ:

+ FactoryX khởi tạo đối tượng của ProductA

+ FactoryY khởi tạo đối tượng của ProductB

* **FactoryManager**: quản lí cấu hình của hệ thống và khởi tạo **AbstractFactory** dựa trên cấu hình đó.

### Mã nguồn minh họa

**Vấn đề**: Bạn cần phát triển hệ thống sản xuất chuỗi cửa hàng ăn nhanh tên là FastFood. Các món bán trong cửa hàng bao gồm: *Hamburger, Sandwich, Fried**chicken*. Tuy nhiên do sự khác nhau về vị trí địa lý và tính phổ biến mà nguyên liệu và giá cả của cùng 1 món sẽ khác nhau ở nhiều cửa hàng. Có tổng cộng 3 cửa hàng: *Đà Nẵng, Hà Nội, Hồ Chí Minh*

Source code: <https://github.com/LichND/Design-Pattern/tree/main/AbstractFactory>

### Hệ quả

* Tuân thủ nguyên tắc Đóng gói (Encapsulation): Hệ thống giấu đi việc khởi tạo các class con. Ngược lại, việc khởi tạo là công việc của các Factory.
* Tuân thủ nguyên tắc Dependency Inversion: Thay vì người dùng trực tiếp khởi tạo các Factory thì lúc này việc khởi tạo được dành cho AbstractFactory.
* Đảm bảo sự đồng nhất giữa các class có cùng hệ.
* Khả năng mở rộng cao.

### Ví dụ thực tế

**Inversion of Control (IoC)**: được ứng dụng để khởi tạo các đối tượng phức tạp trong 1 hệ thống

**Color theme:** việc cài đặt các theme khác nhau cho hệ thống được cài đặt dựa trên Abstract factory. Mỗi các nhóm màu được quy định trong 1 factory cụ thể. Lớp abstract factory sẽ trả về factory con tương ứng và lấy ra được nhóm màu của theme.

### Các mẫu liên quan

**Factory Method**

Mẫu Abstract Factory được thiết kế dựa trên Factory Method. Hay cụ thể hơn, Abstract Factory là 1 super factory khởi tạo các factory con.

**Singleton**

Thường được sử dụng với Singleton để khởi tạo đối tượng duy nhất của các factory con, từ đó giúp giảm tài nguyên của hệ thống.

## Builder

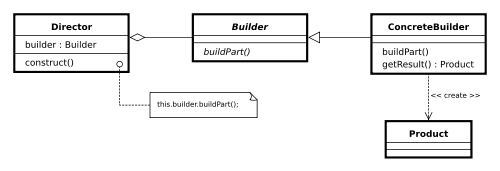
### Tổng quan

* Tên: Builder.
* Phân loại: Abstract Factory thuộc nhóm Creational pattern.
* Bí danh: Monster.
* Mục đích, ý định: tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dụng có thể tạo được các biểu diễn khác nhau

### Khả năng ứng dụng

* Cung cấp một đối tượng để xây dựng một đối tượng phức tạm một cách đơn giản bằng cách sử dụng tiếp cận từng bước. Việc xây dựng này là độc lập.
* Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có quá nhiều thuộc tính.
* Khi Object có quá nhiều thuộc tính sẽ nảy sinh nhiều vấn đề như constructor, get set, tham số tùy chọn, …

### Cấu trúc



Một builder gồm các thành phần cơ bản sau:

* **Product:** đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
* **Builder:** là abstract class hoặc interface khai báo phương thức tạo đối tượng.
* **ConcreteBuilder:** kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
* **Director/Client:** là nơi sẽ gọi tới Builder để tạo ra đối tượng.

Trường hợp đơn giản, chúng ta có thể gộp Builder và ConcreteBuilder thành class nội của Product.

### Mã nguồn minh họa

**Vấn đề**: ở công an ở nước X muốn một database chứa thông tin người dân của mình gồm nhiều trường sao cho một/một số trường có thể rỗng và phải có bắt lỗi các trường (format/true data)​

<https://github.com/LichND/Design-Pattern/tree/main/Builder>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc
* Các class có thể thay thế dễ dàng nhưng vẫn bị ràng buộc đầy đủ chức năng
* Các interface của builder có thể dễ dàng kế thừa, mở rộng
* Thành phần của builder do interface/abstract class quy định

### Ví dụ thực tế

* Tạo object message phức tạp của discord <https://discordjs.guide/popular-topics/embeds.html>
* Tạo các control với nhiều tùy chỉnh trong thiết kế UI

### Các mẫu liên quan

* Factory Pattern: tương tự nhưng cung cấp nhiều quyền kiểm soát hơn đối với quá trình tạo đối tượng. Builder tạo đối tượng step by step trong khi Factory Pattern không rõ cách tạo

## Factory Method

### Tổng quan:

* Tên: Factory Method.
* Phân loại: Factory Method thuộc nhóm Creational pattern.
* Bí danh: Fabrication.
* Mức độ phổ biến: Cao.
* Mục đích, ý định: Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con.

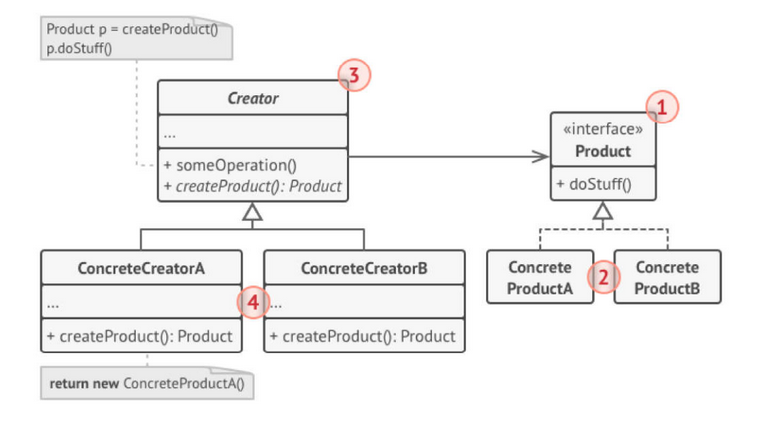
### Khả năng ứng dụng:

- Sử dụng Factory Method khi bạn không biết trước các loại và dependence chính xác của các đối tượng mà code của bạn sẽ hoạt động.

- Sử dụng Factory Method khi bạn muốn cung cấp tới các người dùng một thư viện hoặc framework của mình một cách để mở rộng các thành phần bên trong của nó.

- Sử dụng Factory Method khi bạn muốn tiết kiệm tài nguyên của hệ thống bằng cách tái sử dụng các đối tượng đã có thay vì xây dựng lại chúng.

### Cấu trúc:



***Source: https://refactoring.guru***

– **Product**: định nghĩa interface của các đối tượng mà factory method tạo ra.

– **ConcreteProduct**: thực thi Product interface.

– **Creator:**

+ Xác định factory method trả về đối tượng kiểu Product. Creator cũng có thể định nghĩa một implementation mặc định của factory method trả về một đối tượng ConcreteProduct mặc định.

+ Có thể gọi factory method để tạo ra một đối tượng Product.

**– ConcreteCreator:** override factory method để trả về một đối tượng của một ConcreteProduct.

### Mã nguồn minh họa:

- **Vấn đề:**

+ Một dây chuyền sản xuất bình thường chỉ sản xuất đúng một sản phẩm ly thủy tinh (tenSP, maSP (mã có dạng LTT\_ + số có 2 chữ số)).

+ Công ty muốn dây chuyền này sẽ có thể sản xuất thêm sản phẩm ly sứ. (tenSP, maSP (mã có dạng LS\_ + số có 2 chữ số)).

- https://github.com/LichND/Design-Pattern/tree/main/FactoryMethod

### Hệ quả:

- Với Factory Method, chúng ta có thể thêm nhiều loại product mới vào chương trình mà không làm thay đổi các đoạn code đã có trước đó.

- Mã nguồn có thể trở nên phức tạp hơn vì cần phải sử dụng nhiều lớp con để triển khai mẫu.

### Ví dụ thực tế:

- Ứng dụng trong ADO.NET, trong đó IDbCommand.CreateParameter sử dụng factory method để kết nối những lớp song song.

- Pattern này cũng được sử dụng trong Qt, phương thức QMainWindow :: createPopupMenu là một phương thức gốc được khai báo trong một khuôn khổ có thể được ghi đè trong mã ứng dụng.

- Trong Java một số fatory được sử dụng trong gói javax.xml.parsers. ví dụ. javax.xml.parsers.DocumentBuilderFactory hoặc javax.xml.parsers.SAXParserFactory.

### Các mẫu liên quan:

- Thường thường, các thiết kế bắt đầu với Factory Method (ít phước tạp, nhiều tùy biến, các lớp con nhiều) và sau đó tiến hóa thành Abstract Factory, Prototype hoặc Builder (có nhiều mềm dẻo hơn, nhiều sự phức tạp hơn).

- Abstract Factory có thể được cài đặt cùng với Factory Method.

- Có thể sử dụng Factory Method với Iterator để các collection của các lớp con trả về các loại iterator tương thích với các collection.

- Factory Method là một specialization của Template Method. Đồng thời, Factory Method có thể là một phần trong một Template Method lớn.

## Prototype

### Tổng quan:

* Tên: Prototype.
* Phân loại: Prototype thuộc nhóm Creational pattern.
* Bí danh: Không có.
* Mức độ phổ biến: Trung bình.
* Mục đích, ý định: Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, sau đó tạo mới đối tượng nhờ vào việc sao chép đối tượng mẫu này.

### Khả năng ứng dụng:

- Sử dụng mẫu Prototype khi code không phụ thuộc vào các lớp đối tượng cụ thể mà bạn cần sao chép.

- Sử dụng Prototype khi bạn muốn giảm số lượng clớp con chỉ khác nhau về cách chúng khởi tạo các đối tượng tương ứng. Ai đó có thể đã tạo ra các lớp con này để có thể tạo các đối tượng với một cấu hình cụ thể.

### Cấu trúc:

Graphical user interface, diagram

Description automatically generated

***Source:*** [***https://refactoring.guru***](https://refactoring.guru)

- Interface Prototype: khai báo phương thức clone(). Trong hầu hết các trường hợp, đó là một phương thức clone() duy nhất, các lớp nào cần clone (prototype) thì implement.

- Lớp Concrete Prototype: hiện thực phương thức clone(). Ngoài việc sao chép dữ liệu của đối tượng gốc sang bản sao, phương pháp này cũng có thể xử lý một số trường hợp khác của quá trình sao chép.

- Client: có thể tạo một bản sao của bất kỳ đối tượng nào tuân theo interface Prototype.

### Mã nguồn minh họa

- Dưới đây là một mã nguồn minh hoạ Java thực hiện mẫu Prototype, trong đó class cần clone 3 thuộc tính là ho, tenLot do đó đã implement interface Cloneable.

- https://github.com/LichND/Design-Pattern/tree/main/Prototype

### Hệ quả:

- Việc clone các object phức tạp có chứa các tham chiếu cụ thể cần phải lựa chọn giữa việc có copy các tham chiếu hay không (deep copy versus shallow copy), nếu có thì sẽ phải thực hiện một số thủ thuật.

### Ví dụ thực tế:

- Với các ngôn ngữ lập trình khác nhau đã cung cấp các interface khác nhau nhằm thực hiện tạo class Prototype.

+ Java: Cloneable.

+ C#: ICloneable.

- Mẫu này trong thực tế được sử dụng ở một số loại phần mềm đặc biệt như đồ hoạ máy tính, CAD, hệ thống địa chất, game.

### Các mẫu liên quan:

- Thường thường, các thiết kế bắt đầu với Factory Method (ít phước tạp, nhiều tùy biến, các lớp con nhiều) và sau đó tiến hóa thành Abstract Factory, Prototype hoặc Builder (có nhiều mềm dẻo hơn, nhiều sự phức tạp hơn).

- Abstract Factory có thể sử dụng Prototype tạo ra các phương thức cho các class của nó.

- Prototype có thể được sử dụng khi muốn lưu bản sao của Commands vào lịch sử.

- Cac thiết kế tốn tài nguyên sử dụng Composite và Decorator có thể sử dụng Prototype để copy các object phức tạp thay vì tái cấu trúc lại chúng.

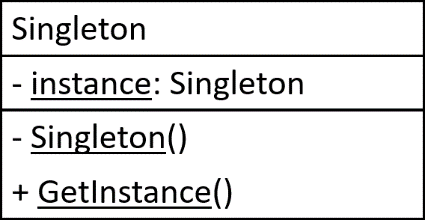
- Đôi khi Prototype có thể là một sự thay thế đơn giản cho Memento.

## Singleton

### Tổng quan

* Tên: Singleton.
* Phân loại: Singleton thuộc nhóm Creational pattern.
* Mục đích, ý định: Cài đặt lớp mà chỉ có một thể hiện (đối tượng) duy nhất

### Cấu trúc



Có nhiều cách để implement Singleton nhưng phải bảo đảm:

* Private constructor để hạn chế truy cập từ class bên ngoài.
* Đặt private static final variable đảm bảo biến chỉ được khởi tạo trong class.
* Có một method public static để return instance được khởi tạo ở trên.

### Mã nguồn minh họa

Như đã nói ở trên, có nhiều cách implement Singleton tùy vào từng trường hợp. Cụ thể:

* Eager Initialization: <https://github.com/LichND/Design-Pattern/blob/main/Singleton/DemoSingleton/EagerInitialization.cs>
* Lazy Initialization: <https://github.com/LichND/Design-Pattern/blob/main/Singleton/DemoSingleton/LazyInitialization.cs>
* Thread Safe Singleton: <https://github.com/LichND/Design-Pattern/blob/main/Singleton/DemoSingleton/ThreadSafeSingleton.cs>
* Bill Pugh Implementation: <https://github.com/LichND/Design-Pattern/blob/main/Singleton/DemoSingleton/BillPughImplementation.cs>

### Hệ quả

* Singleton khiến bất cứ class nào cũng phải gánh việc quản lý khởi tạo chính nó.
* Các class hoàn toàn có thể thay thế được 1 cách dễ dàng vì việc khởi tạo là trách nhiệm của class cha hoặc chính bản thân nó.

### Ví dụ thực tế

Có rất nhiều trường hợp cần sử dụng Singleton, ví dụ:

* Dùng làm cache
* Logger
* Database
* Pool

## Adapter

### Tổng quan

* Tên: Adapter.
* Phân loại: Structual pattern.
* Mức độ phổ biến: Cao
* Mục đích, ý định: Adapter hoạt động như 1 cầu nối giữa 2 interface không tương thích với nhau.

Có 2 loại mẫu adapter: **Object adapter** và **Class adapter.**

### Khả năng ứng dụng

* Khi hệ thống cần sử dụng các class ở bên thứ 3 nhưng interface lại không tương thích.
* Khi cần mở rộng chức năng của một số class có sẵn trong hệ thống nhưng lại không thể đưa các chức năng đó vào class cha.

### Cấu trúc

****Object Adapter**

Adapter gồm 3 thành phần chính:

* **Client:** bao gồm các thành phần của client dùng để giao tiếp trực tiếp với hệ thống. Cụ thể trong thiết kế gồm 1 interface **IClientInterface** và 1 class **ConcreteClient** kế thừa từ interface trên
* **Service:** các thành phần của service cần sử dụng trong client. Cụ thể ở đây có 1 interface **I3rdService** và 2 class Service kế thừa từ interface trên.
* **Adapter**: Kế thừa từ **IClientInterface** và có tham chiếu đến **I3rdService**. Bản chất adapter lúc này như 1 client nhưng được tích hợp thêm service vào trong class.

**Class Adapter**

Class Adapter cũng bao gồm 3 thành phần giống như Object Adapter. Tuy nhiên lúc này Adapter kế thừa từ class **3rdService**.

*Hạn chế:*

* Ngôn ngữ phải hỗ trợ đa kế thừa (multiple classes inheritance)
* **3rdService** phải là 1 class (tăng mức độ phụ thuộc của adapter)

### Mã nguồn minh họa

Source code: <https://github.com/LichND/Design-Pattern/tree/main/Adapter>

### Hệ quả

* Không cần mở rộng interface cho cùng 1 chức năng hoặc cho các chức năng mà hệ thống sẽ không dùng (**Interface Segragation**)
* Đảm bảo tính ổn định của các interface gốc của hệ thống (**Open/Close principle**).
* Đảm bảo các thành phần mở rộng không phụ thuộc vào các lớp hệ thống.

### Ví dụ thực tế

**Socket adapter**

Socket là 1 mô hình giao tiếp realtime giữa client và server trên các channel với kết nối ổn định. Tuy nhiên, khá với bản chất của RESTful API, Socket lại không phải là stateless. Mặc định các client kết nối socket với nhau trên 1 process mới có thể giao tiếp với nhau.

Do đó, socket adapter là 1 mô hình thiết kế giúp mở rộng việc giao tiếp giữa các socket trên nhiều process khác nhau. Socket adapter lúc này giữ tham chiếu đến thể hiện của Server và các thông điệp gửi và nhận của client sẽ được gửi qua Server.

## Bridge

### Tổng quan:

* Tên: Bridge.
* Phân loại: Structual pattern.
* Mức độ phổ biến: Trung bình.
* Mục đích, ý định: Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt ; mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau.

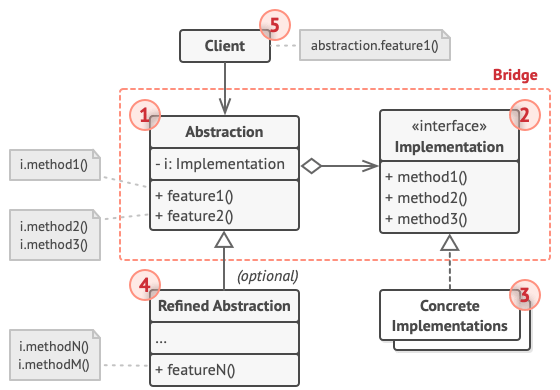
### Khả năng ứng dụng:

- Sử dụng mẫu Bridge khi muốn phân chia và tổ chức một lớp monolithic có một số biến thể của một số phương thức nhất định.

- Sử dụng Bridge khi muốn mở rộng các class ở các phương thức của nó một cách độc lập.

- Sử dụng Bridge để có thể chuyển đổi các implementation ngay trong runtime.

### Cấu trúc:



***Source: https://refactoring.guru***

- Abstraction cung cấp một logic high-level cho quá trình code. Các implementation cẩn phải thực hiện các công việc cụ thể từ logic đó.

- Các interface Implementation khai báo interface cho tất cả các Concrete Implementation. Abstraction chỉ có thể tương tác với các đối tượng implementation tại đây.

- Các Concrete Implementation chứa các đoạn code thực hiện hoá các phương thức theo các cách khác nhau cho cùng một phương thức.

### Mã nguồn minh họa:

- Trong ví dụ bên dưới, minh hoạ quá trình tạo ra một website gồm 2 giai đoạn là thiết kế và code, nếu gộp chung cả hai thành 1 class duy nhất sẽ không phù hợp với thực tiễn. Đồng thời các class thực hiện từng loại công việc cũng có các công việc cụ thể khác nhau tuy nhiên quá trình tạo ra sản phẩm (Header, Footer) có thể khác nhau.

- https://github.com/LichND/Design-Pattern/tree/main/Bridge/src

### Hệ quả

\* Ưu điểm:

- Có thể tạo ra các lớp với thành phần trừu tượng và thành phần thực thi độc lập với nhau.

- Người dùng sẽ sử dụng code ở high-level, che dấu được các chi tiết thực thi.

- Open/Closed Principle: *có thể tạo abstractions* và các *implementation* độc lập với nhau*.*

- Single Responsibility Principle*:* Tập trung các logic ở high-level (abstraction)và các chi tiết ở low-level (implementation).

\* Nhược điểm:

- Nếu gặp phải các lớp có tính gắn kết cao, thì rất dễ để tạo ra các class phức tạp.

### Ví dụ thực tế:

- ODBC, API này chứa interface trong Bridge design pattern – các class implement API là các ODBC driver.

### Các mẫu liên quan:

- Bridge, State, Strategy (và một phần nhỏ của Adapter) có cấu trúc khá giống nhau. Các pattern này dựa vào các thành phần sẽ được delegate tới các object khác.

- Có thể kết hợp Builder với Bridge: Director class thực hiện vai trò Abstraction, còn các builder sẽ thực hiện các implementation.

## Composite

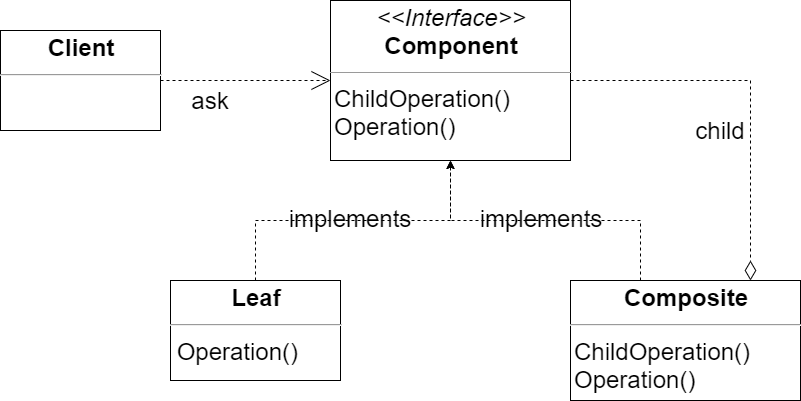
### Tổng quan

* Tên: Composite.
* Phân loại: Composite thuộc nhóm Structural Pattern.
* Mục đích, ý định: tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau

### Khả năng ứng dụng

* Composite sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp.
* Nó tạo một lớp chứa nhóm đối tượng của riêng nó, cung cấp các cách để sửa đổi nhóm của cùng 1 object.
* Cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng

### Cấu trúc



Một Composite Pattern bao gồm các thành phần cơ bản sau:

* **Component**: là interface hoặc abstract class quy định các base method
* **Leaf**: là lớp hiện thực Component mà không có con (Leaf)
* **Composite**: chứa các Leaf và hiện thực base để ủy nhiệm cho Leaf xử lý
* **Client**: làm việc thông qua Component

### Mã nguồn minh họa

**Vấn đề**: Công ty X có một dây truyền xản xuất hàng loạt các sản phẩm với các bước như sau:

* Mua nguyên liệu
* Xử lý nguyên liệu
* Đóng gói

Ứng với mỗi sản phẩm lại có thời gian thực hiện, chi phí và các quy trình bên trong khác nhau. Viết phần mềm thống kê quản lý việc xản xuất trên.

<https://github.com/LichND/Design-Pattern/tree/main/Composite>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc
* Các class tuy có thể thay thế dễ dàng nhưng class Leaf vì không implement các childOpration (để trống) nên vi phạm Liskov substitution principle.
* Các interface của Composite có thể dễ dàng kế thừa, mở rộng.
* Thành phần của  Composite do interface/abstract class quy định.

### Ví dụ thực tế

Trong thực tế ta có thể dễ dàng thấy ứng dụng của composite được dùng ở khắp mọi nơi:

* Xây dựng ứng dụng
* Quản lý đối tượng
* Mô hình xử lý công việc bằng cách giao cho đối tượng chuyên trách

## Decorator

### Tổng quan

* Tên: Decorator.
* Phân loại: Decorator thuộc nhóm Structural Pattern.
* Mục đích, ý định: giúp mở rộng cài đặt của 1 lớp mà không thay đổi cấu trúc của nó. Lớp mở rộng được xem như 1 wrapper của lớp đã có sẵn.

### Khả năng ứng dụng

* Sử dụng khi cần bổ sung thêm các hành vi cho đối tượng tại thời điểm thực thi mà không cần phải thay đổi code.
* Sử dụng khi cảm thấy việc kế thừa là bất khả thi hoặc sai lệch bản chất của đối tượng.

### Cấu trúc

Gồm 2 thành phần:

**Decoratee**: các thành phần của hệ thống cần được mở rộng chức năng. Cụ thể bao gồm 1 interface **IDecoratee** và các class con tương ứng.

**Decorator**: các thành phần mở rộng chức năng cho **Decoratee**. Trong đó:

* **BaseDecorator**: là 1 lớp trừu tượng implement **IDecoratee** và có 1 tham chiếu đến **IDecoratee.**
* Các class con kế thừa từ BaseDecorator và cài đặt các chức năng mở rộng cho Decoratee.

### Mã nguồn minh họa

Bạn bước vào tiệm bánh canh, khi gọi món bạn có thể có nhiều yêu cầu khác nhau về gia vị và thành phần. Trong đó bánh canh là món chính, gia vị và các món ăn thêm như chả cá, hành ngò, da heo, tiêu, … Tùy vào khẩu vị mà bạn có thể yêu cầu khác nhau về món ăn như bánh canh không hành, hoặc không cho tiêu vào, … Từ đó có thể thấy chỉ một món ăn chính và vài thành phần phụ đã tạo ra rất nhiều biến thể của món ăn. Tuy nhiên, với việc áp dụng mẫu Decorator, chúng ta có thể quản lí chúng một cách dễ dàng.

Source code: <https://github.com/LichND/Design-Pattern/tree/main/Decorator>

### Hệ quả

* Đơn giản hóa hệ thống: Các lớp này đều có cùng 1 hành vi. Do đó việc quản lí dựa trên Decorator sẽ đơn giản hóa quá trình quản lí cũng như việc gọi thực thi chương trình.
* Khả năng mở rộng cao, linh hoạt.

Nhược điểm:

* Các thành phần bắt buộc phải khởi tạo từ đầu và không thể thay đổi trong quá trình hoạt động.
* Không thể áp dụng với các thành phần có nhiều chức năng hoặc quá phức tạp hoặc phụ thuộc lẫn nhau.

### Ví dụ thực tế

**Được sử dụng nhiều trong việc mở rộng chức năng của lớp**

Trong rất nhiều trường hợp, chúng ta cần thay đổi chức năng của 1 hàm nhưng lại không nắm rõ những yêu cầu có thể được đưa ra trong tương lai. Nếu chúng ta mở rộng bằng việc thêm hàm mới thì class sẽ phình to 1 cách không cần thiết (vì các hàm đều có chung 1 biểu hiện). Ngược lại, việc sử dụng Decorator cho phép chúng ta mở rộng hàm mà không cần can thiệp vào lớp cơ sở. Khi có sự thay đổi về yêu cầu thì chúng ta chỉ cần đơn giản thay lớp Decorator bằng lớp Decorator khác.

### Các mẫu liên quan

**Prototype**

Decorator có cấu trúc nhiều lớp phức tạp nên có thể kết hợp với Prototype để đơn giản hóa việc sao chép.

**Adapter**

Tuy cả adapter và decorator có cấu trúc giống nhau lại có mục đích sử dụng hoàn toàn khác nhau. Adapter giúp chúng ta tích hợp các lớp bên thứ 3 vào trong hệ thống, trong khi đó, Decorator giúp mở rộng phạm vi hoạt động của 1 hàm bên trong class cơ sở.

## Facade

### Tổng quan:

* Tên: Facade.
* Phân loại: Structual pattern.
* Mức độ phổ biến: Cao.
* Mục đích, ý định: Cung cấp một interface duy nhất cho một tập hợp các interface trong một subsystem. Nó định nghĩa một interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.

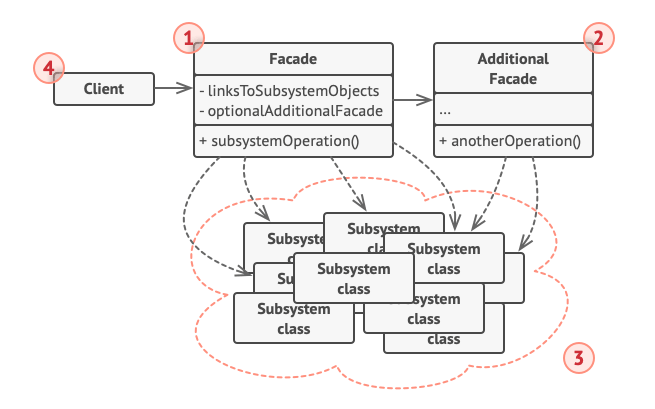
### Khả năng ứng dụng:

- Sử dụng Facade pattern khi cần một interface đơn giản để thao tác với một hệ thống phức tạp.

- Sử dụng Facade khi muốn cấu trúc một hệ thống con thành các lớp.

- Khi muốn che dấu sự phức tạp của hệ thống với người dùng.

### Cấu trúc:



***Source: https://refactoring.guru***

- Facade: là nơi có thể truy cập các lớp của hệ thống subsystem.

- Additional Facade: Một facade bổ xung để tránh làm cho Facade chính phức tạp.

- Subsystem.

### Mã nguồn minh họa:

- Ví dụ bên dưới mô tả hành động của một sinh viên tương tác với hệ thống sách thông qua các phương thức đơn giản thực sự cần cho sinh viên mà không quan tâm tới bên trong phương thức đó chứa đựng những xử lý phức tạp nào.

- https://github.com/LichND/Design-Pattern/tree/main/Facade

### Hệ quả:

\*Ưu điểm:

- Có thể code nghiệp vụ một cách độc lập với các subsystem.

- Làm giảm sự coupling giữa các class trong subsystem.

- Tăng khả năng hiểu và tái sử dụng của code.

\*Nhược điểm: Dễ tạo ra một class có liên quan tới quá nhiều object (god class) trong một project.

### Ví dụ thực tế:

- Ghi log trong .NET Framework.

### Các mẫu liên quan:

- Facade định nghĩa interface mới từ các object đã có, Adapter thì cố tái sử dụng các interface có sẵn.

- Abstract Factory có thể sử dụng thay cho Facade nếu mục đích sử dụng chỉ là che dấu sự phức tạp của hệ thống tới người dùng.

- Flyweight giúp tạo ra các object nhỏ, Facade chỉ cách để tạo ra ít object để tương tác với cả một hệ thống lớn.

- Một class Facade có thể chuyển về dạng Singleton nếu chỉ có một đối tượng Facade duy nhất được sử dụng nhiều lần.

- Facade khá giống với Proxy cả hai đều tạo ra một bộ đệm để thao tác với một hệ thống phức tạp.

## Flyweigth

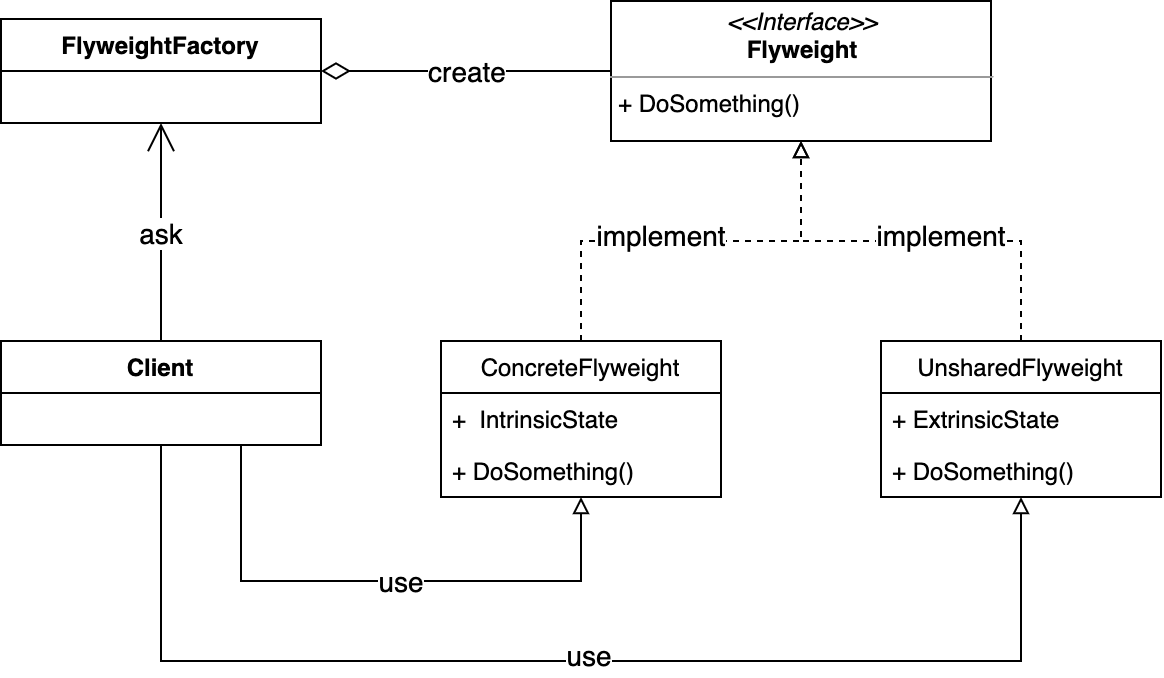
### Tổng quan

* Tên: Flyweight.
* Phân loại: Flyweight thuộc nhóm Structural Pattern.
* Bí danh: Poids mouche.
* Mục đích, ý định: sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng "cỡ nhỏ"

### Khả năng ứng dụng

* Flyweight Pattern được sử dụng khi chúng ta cần sử dụng một số lượng lớn các đối tượng của 1 lớp nào đó.
* Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra và hủy bỏ liên tục có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp.
* Flyweight Pattern được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng có thể tái sử dụng.
* Flyweight object là immutable, nghĩa là không thể thay đổi khi nó đã được khởi tạo.

### Cấu trúc



Các thành phần trong mẫu thiết kế Flyweight:

* **Flyweight**: là một interface/ abstract class, định nghĩa các các thành phần của một đối tượng.
* **ConcreteFlyweight**: triển khai các phương thức đã được định nghĩa trong Flyweight. Việc triển khai này phải thực hiện các khả năng của trạng thái nội tại. Đó là dữ liệu phải không thể thay đổi (unchangeable) và có thể chia sẻ (shareable).
* **UnsharedFlyweight**: dù mẫu thiết kế Flyweight cho phép chia sẻ thông tin, nhưng vẫn có thể tạo ra các thể hiện không được chia sẻ (not shared). Trong những trường hợp này, thông tin của các đối tượng có thể là stateful. Do đó client phải có tránh nhiệm truyền dữ liệu liên quan thông qua argument.
* **FlyweightFactory (Cache):** lớp này có thể là một Factory Pattern được sử dụng để giữ tham chiếu đến đối tượng Flyweight đã được tạo ra. Nó cung cấp một phương thức để truy cập đối tượng Flyweight được chia sẽ. FlyweightFactory bao gồm một Pool (có thể là HashMap, không cho phép bên ngoài truy cập vào) để lưu trữ đối tượng Flyweight trong bộ nhớ. Nó sẽ trả về đối tượng Flyweight đã tồn tại khi được yêu cầu từ Client hoặc tạo mới nếu không tồn tại.
* **Client**: sử dụng FlyweightFactory để khởi tạo đối tượng Flyweight.

### Mã nguồn minh họa

**Vấn đề**: Một nhà hành X có một menu thức ăn chung để khách hàng gọi món. Menu này là giống nhau. Tuy nhiêu món ăn có thể thay đổi tùy theo sở thích của khách hàng. Quá trình xem menu và đặt món ăn của khách hành là diễn ra liên tục. Viết chương trình mô phỏng quá trình này

<https://github.com/LichND/Design-Pattern/tree/main/Flyweight>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc
* Các class con có thể dễ dàng thay thế class cha mà không làm ảnh hưởng đến chương trình
* Các interface của Flyweight có thể dễ dàng kế thừa, mở rộng
* Thành phần của Flyweight do interface/abstract class quy định
* Giảm tầng xuất tạo ra vào mất đi của object
* Tiết kiệm bộ nhớ lưu trữ
* Tăng performance
* Dễ dàng kết hợp với nhiều design khác

### Ví dụ thực tế

Flyweight được dùng nhiều nhất ở hai dạng chính trong gần như toàn bộ các chương trình hiện nay

* Pool Object: Particular, Spawn machine, …
* Cache: Database, Memcache, …

## Proxy

### Tổng quan

* Tên: Proxy.
* Phân loại: Proxy thuộc nhóm Structural Pattern.
* Bí danh: Procuration.
* Mục đích, ý định: Điều khiển một cách gián tiếp việc truy xuất một đối tượng thông qua một đối tượng khác được ủy nhiệm

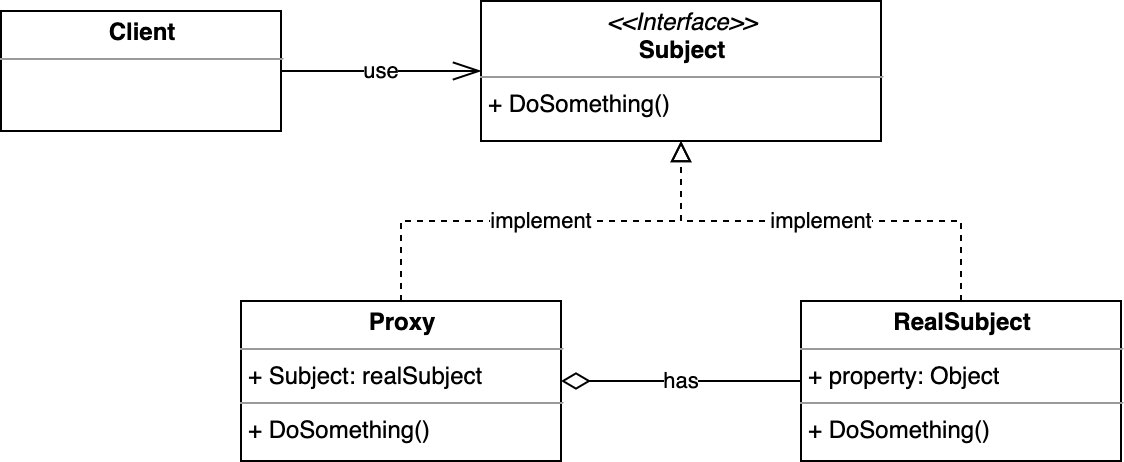
### Khả năng ứng dụng

* Tạo đối tượng trung gian khi có yêu cầu để giảm bớt tài nguyên cần thiết để tạo hoàn chỉnh đối tượng chính
* Bảo vệ đối tượng chính khỏi dữ liệu nguy hại hay người dùng không có quyền
* Bảo vệ dữ liệu của đối tượng chính
* Tái sử dụng thông tin của đối tượng chính
* Kiểm soát số lượng tham chiếu đến đối tượng chính
* Trách xung đột khi có nhiều lệnh ghi vào đối tượng chính

### Cấu trúc

Tùy vào nhu cầu cũng như mục đích sử dụng mà Proxy pattern có cài đặt đôi chút khác biệt, tuy nhiên vẫn phải có các đặc điểm chung sau:

* Cung cấp mức truy cập gián tiếp vào một đối tượng.
* Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó.
* Cả Proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện.



Các thành phần tham gia vào mẫu Proxy Pattern:

* **Subject**: là một interface định nghĩa các phương thực để giao tiếp với client. Đối tượng này xác định giao diện chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject mong đợi.
* **Proxy**: là một class sẽ thực hiện các bước kiểm tra và gọi tới đối thật để thực hiện các thao tác sau khi kiểm tra. Nó duy trì một tham chiếu đến RealSubject để Proxy có thể truy cập nó. Nó cũng thực hiện các giao diện tương tự như RealSubject để Proxy có thể được sử dụng thay cho RealSubject. Proxy cũng điều khiển truy cập vào RealSubject và có thể tạo hoặc xóa đối tượng này.
* **RealSubject**: là một class service sẽ thực hiện các thao tác thực sự. Đây là đối tượng chính mà proxy đại diện.
* **Client**: Đối tượng cần sử dụng RealSubject nhưng thông qua Proxy.

### Mã nguồn minh họa

Đây là một ví dụ đơn giản cho trường hợp dùng Proxy để kiểm tra quyền người dùng: <https://github.com/LichND/Design-Pattern/tree/main/Proxy>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc.
* Các class con dễ dàng thay đổi class cha mà không làm ảnh hưởng đến chương trình.
* Các interface của Proxy có thể dễ dàng kế thừa, mở rộng.
* Thành phần của Proxy do interface/abstract class quy định.

### Ví dụ thực tế

Trong thực tế có rất nhiều lý do để tạo proxy. Ví dụ:

* Web proxy
* Remote proxy
* Firewall Proxy
* Cache Proxy
* Copy-On-Write Proxy
* …

### Các mẫu liên quan

* Cấu trúc của Proxy Pattern và Decorator Pattern là tương tự nhau. Hai Pattern này đều Wrap một đối tượng thực bên trong nó. Sự khác nhau thật sự giữa Proxy Pattern và Decorator Pattern nằm ở mục đích sử dụng.
* Với Decorator Pattern, người sử dụng sẽ nhắm tới mục tiêu là có thể thêm tính năng động vào một đối tượng có trước, trong khi đó Proxy Pattern cho phép ta tạo ra một đại diện cho một đối tượng khác.

## Chain or Responsibility

### Tổng quan:

- Tên: Chain of Responsibility.

- Phân loại: Chain of Responsibility thuộc nhóm Behavior Pattern.

- Bí danh: Chaine de responsabilités.

- Mục đích, ý định: Khắc phục việc ghép cặp giữa các bộ gởi và bộ nhận thông điệp. Với mẫu thiết kế này, các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó.Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có có cơ hội xử lý request. Liên kết các đối tượng nhận request thành 1 dây chuyền rồi "pass" request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.

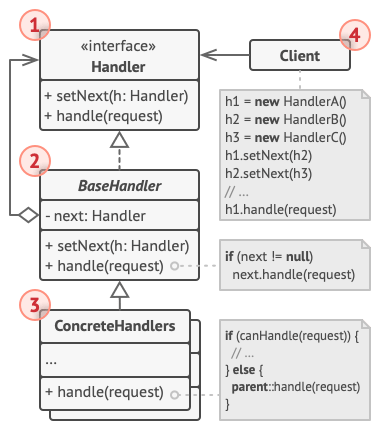
### Khả năng ứng dụng:

- Sử dụng Chain of Responsibility khi chương trình dự kiến sẽ xử lý nhiều loại yêu cầu khác nhau, nhưng chưa biết trước loại yêu cầu và trình tự của chúng.

- Sử dụng mẫu này khi cần thực thi một số công việc theo thứ tự cụ thể.

- Với trình tự các công việc sẽ bị thay đổi trong lúc runtime cũng có thể sử dụng mẫu này.

### Cấu trúc:



***Source: https://refactoring.guru***

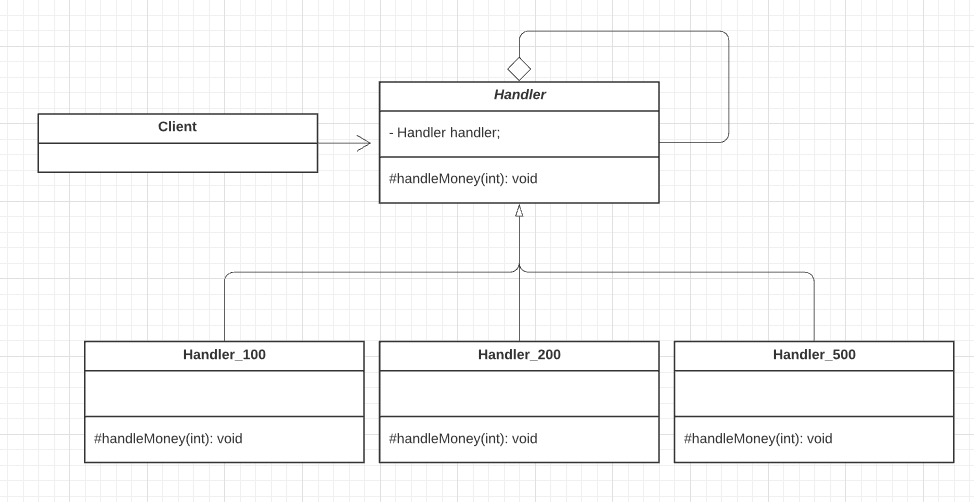
- **Handler** khai báo 1 interface chung cho tất cả các ConcreteHandler. Trong interface này thường chứa 1 phương thức để xử lý request và đôi khi chứa các phương thức để cài đặt phương thức tiếp theo trong chuỗi.

- **Base Handler** là một class optional chứa các mẫu code chung cho tất cả các class.

**- Receiver/ConreteHandler** implement từ *Handler* và handler method, xử lý được 2 trường hợp: nếu Receiver là nơi xử lý request/nếu nó không xử lý mà pass request lại cho các Receiver khác.

### Mã nguồn minh họa:

- Trong ví dụ bên dưới xử lý một yêu cầu đầu vào là một số tiền, các loại tiền khác nhau được xử lý theo thứ tự: 500, 200 và 100. Nếu số tiền lớn hơn hoặc bằng 500, thì classHandler 500 sẽ được gọi, tương tự với 200 và 100.



- https://github.com/LichND/Design-Pattern/tree/main/ChainOfResponsibility

### Hệ quả:

- Client có thể kích hoạt bất kỳ handler nào trong chuỗi / một số request có thể không cần đến hết được chain / một số request không được xử lý dù đã qua hết các handler trong chain.

- Một số ưu điểm:

+ Có thể sắp xếp được thứ tự các handler.

+ Thoả mãn Single Responsibility Principle.

+ Thoả mãn Open/Closed Principle**.**

### Ví dụ thực tế:

- Cocoa and Cocoa Touch framework, được sử dụng cho OS X và các ứng dụng iOS, sử dụng chain-of-responsibility pattern cho việc xử lý các các event.

- .NET Framework: Windows event model.

A close up of a sign

Description automatically generated

***Source: https://refactoring.guru***

- Trong Java:

+ javax.servlet.Filter#doFilter()

+ java.util.logging.Logger#log()

### Các mẫu liên quan:

- Có thể kết hợp với Composite partten: khi một leaf component nhận 1 request, nó sẽ pass request đó đi lên các parent component.

- Có class diagram gần giống với Decorator, tuy nhiên nó không cho thay đổi luồng đi của các request.

- Việc xử lý các sự kiện trong Chain of Responsibility có thể thực hiện bằng Commands.

## Command

### Tổng quan

* Tên: Command.
* Phân loại: Command thuộc nhóm Behavioral Pattern.
* Mục đích, ý định:
* Đưa việc xử lí logic và các thông tin cần thiết vào trong thành phần gọi là **Command**.
* Command sau đó sẽ được gọi để thực thi công việc. Thành phần gọi tới Command để xử lý được gọi là **Invoker**.

### Khả năng ứng dụng

* Cần 1 giải pháp quản lí các xử lí logic như các object riêng biệt (thêm, loại bỏ, thực thi, …).
* Khi cần linh động về thời gian thực thi (xử lí request theo queue, schedule, …).
* Khi cần implement các thao tác reverse, undo.

### Cấu trúc

Bao gồm 3 thành phần chính:

**Command**: là 1 wrapper đối với phần lệnh cần thực thi. Cụ thể ở đây là interface **Command** và 2 class **CommandA** và **CommandB.**

Có 2 loại command:

Normal command: các command được thực thi độc lập mà không cần tham số nào từ bên ngoài.

Param command: các command phụ thuộc vào tham số để thực thi.

Mỗi command đều có hàm **execute**. Hàm này chịu trách nhiệm thực thi công việc được yêu cầu.

**Receiver:** đối tượng có chức năng cần được thực thi bên trong command. Cụ thể ở trong thiết kế là hàm **doSomeThing**. **Command** có giữ tham chiếu đến **Receiver** để có thể gọi hàm này.

**Invoker:** là 1 class quản lí và thực thi command. Invoker có tham chiếu đến command.

### Hệ quả

* Tuân thủ **Single Responsibility.** Bởi vì chỉ có trách nhiệm duy nhất là gọi hàm cần thực thi, do đó, việc bổ sung các command mới không ảnh hưởng nhiều đến các command trước đó.
* Tuân thủ **Open/Closed Principle.**
* Phân tầng giữa các thành phần riêng biệt trong hệ thống **( Separation of Concerns).** Lúc này Client sẽ gọi thực thi các hàm của hệ thống thông qua các command.

### Ví dụ thực tế

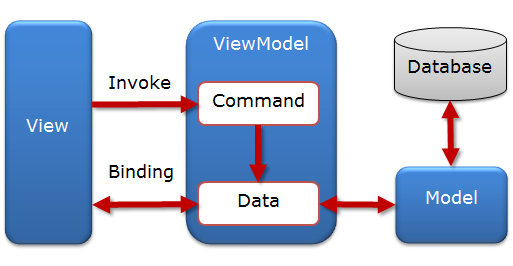
**Reverse Operations**

Gần như các hệ thống làm việc với dữ liệu đều có chức năng Reverse (hoặc Undo) để khôi phục lại trạng thái của hệ thống trước khi thực hiện 1 thao tác nào đó. Tuy nhiên để làm được điều này cần phải có 1 phương thức hiệu quả để quản lí các thao tác cũng như trạng thái của hệ thống.

Trong sơ đồ trên, các command được quản lí thông qua class **CommandHistory**. Class này có 2 hàm push và pop dùng để lưu lại thao tác mới và lấy ra thao tác mới nhất khi cần reverse.

Trạng thái của chương trình được lưu giữ bên trong **UndoCommand**. Do đó, khi lệnh undo được gọi, chương trình có thể trả về trạng thái trước đó.

**Mô hình MVC, MVVM:**

Các mô hình thiết kế hệ thống MVC và MVVM đều có sự phân lớp giữa các thành phần giao diện, logic và dữ liệu. Các thành phần giao diện không tương tác trực tiếp với phần xử lý business mà thông qua các command để thực thi.

### Các mẫu liên quan

**Observer**:

Mẫu observer dùng hiện thực hóa mô hình event-based system mà tại đó, các chức năng được gọi khi có sự kiện diễn ra.

Khi kết hợp với Command, các subcriber có thể giao tiếp với hệ thống thông qua các Command thay vì gọi trực tiếp phần xử lí bên trong của hệ thống.

**Prototype**

Có thể kết hợp với prototype để đơn giản hóa việc tạo các bản copy của command.

**Visitor**

Visitor và Command có chung mục đích sử dụng.

Visitor linh động hơn trong việc xử lý các object của các class khác nhau và có tham số khác nhau.

## Interpreter

### Tổng quan

* Tên: Interpreter.
* Phân loại: Intepreter thuộc nhóm Behavioral Pattern.
* Mục đích, ý định: Dùng để định nghĩa các cú pháp (syntax) cho 1 ngôn ngữ và cách diễn giải các syntax đó. Cấu trúc này được gọi là **Abtract Syntax Tree (AST).**

### Khả năng ứng dụng

* Khi cần định nghĩa (hoặc biên dịch) một loại ngôn ngữ với cú pháp đơn giản (SQL Parsing, XML, các biểu thức toán, …).

### Cấu trúc

Mô hình gồm 2 thành phần:

**Expression**: định nghĩa các syntax và cách diển giải các syntax đó. Có 2 loại expression

* Terminal expression: biểu thức đơn có khả năng tự định nghĩa.
* Nonterminal expression: biểu thức có cấu trúc phức (tạo nên từ các biểu thức đơn hoặc phức khác).

**Context**: đầu vào của expression, chứa thông tin mà expression cần diễn giải,

### Mã nguồn minh họa

Thiết kế chương trình để chuyển từ chữ số La Mã sang chữ số thường.

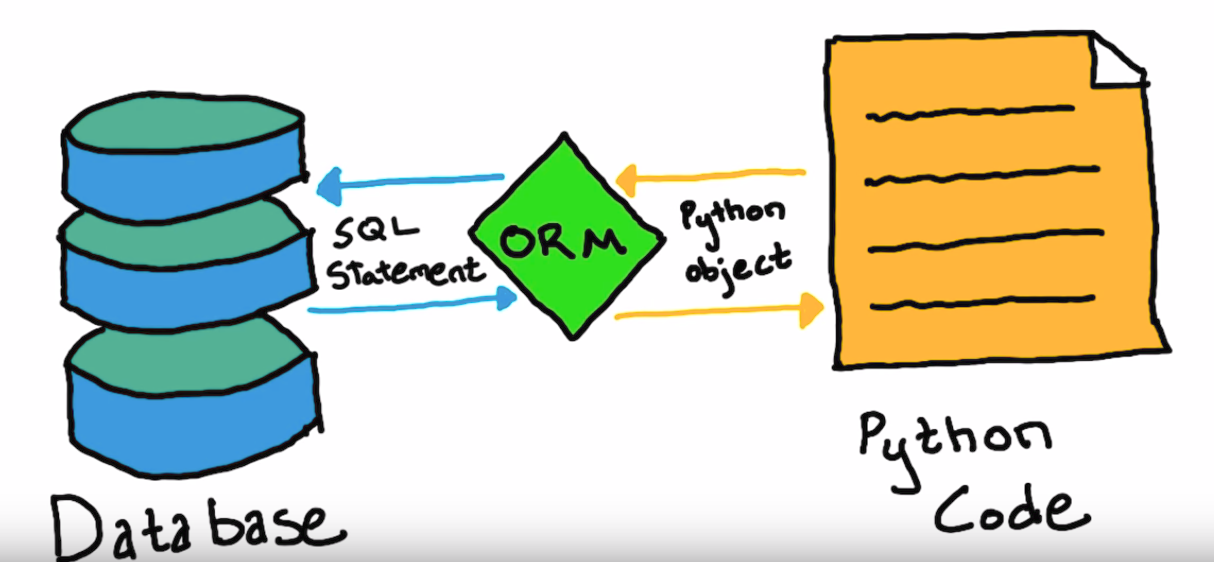
Source code: <https://github.com/LichND/Design-Pattern/tree/main/Interpreter>

### Hệ quả

* Tuân thủ nguyên tắc **Open/Closed principle.**

### Ví dụ thực tế

**ORM (object relational mapping)**

ORM là mô hình nhằm diễn giải các đối tượng, câu lệnh ở 1 hệ thống khác (như SQL, MySQL, Cloud, …) thành các hàm có thể thực thi được bên trong chương trình, qua đó giúp đơn giản hóa việc giao tiếp và đồng bộ giữa các hệ thống với nhau.

### Các mẫu liên quan

**Decorator**

* Kiểu cấu trúc là stack.
* giúp mở rộng khả năng hoạt động nhưng vẫn giữ cấu trúc đơn giản.

**Composite**:

* Kiểu cấu trúc là cây.
* Được sử dụng khi cần biểu diễn các đối tượng cấu trúc dạng phân cấp.

**Interpreter**

* Kiểu cấu trúc là cây.
* Diễn giải cú pháp (syntax) của ngôn ngữ.

## Iterator

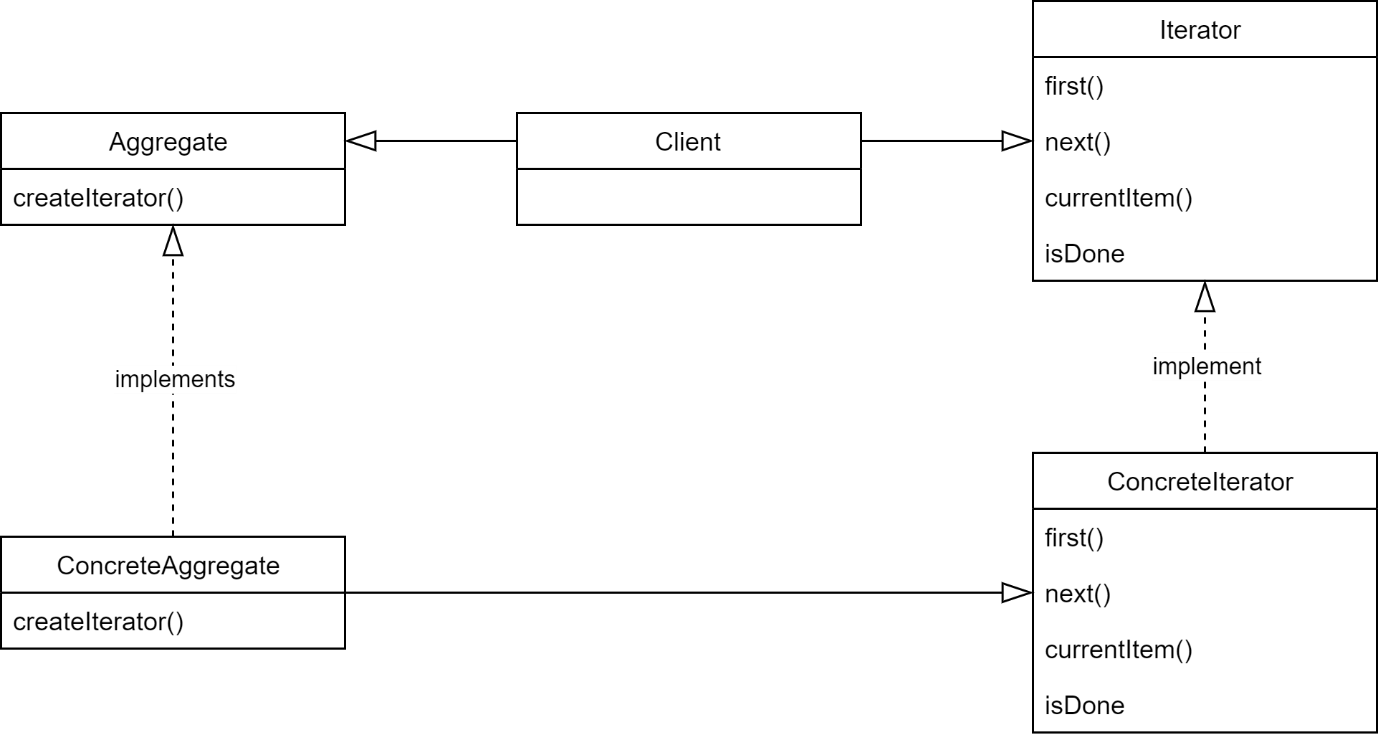
### Tổng quan

* Tên: Iterrator.
* Phân loại: Iterrator thuộc nhóm Behavior Pattern.
* Bí danh: Itérateur.
* Mục đích, ý định: Truy xuất các phần tử của đối tượng dạng tập hợp mà không phụ thuộc và biểu diễn bên trong của các phần tử

### Khả năng ứng dụng

* Cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.
* Tách biệt trách nhiệm của bản thân lớp tập hợp với việc duyệt.

### Cấu trúc



Các thành phần tham gia mẫu Iterator:

* **Aggregate**: một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.
* **ConcreteAggregate**: cài đặt các phương thức của Aggregate, nó cài đặt interface tạo Iterator để trả về một thể hiện của ConcreteIterator thích hợp.
* **Iterator**: là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử.
* **ConcreteIterator**: cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử.
* **Client**: đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp do ConcreteIterator quyết định.

### Mã nguồn minh họa

Đây là một ví dụ đơn giả của Iterator. Trong thực thế hầu hết các ngôn ngữ lập trình đều đã implement Iterator để phục vụ cho việc indexing với performent cao nhất

<https://github.com/LichND/Design-Pattern/tree/main/Iterator>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc
* Các class con dễ dàng thay đổi class cha mà không làm ảnh hưởng đến chương trình
* Các interface có thể dễ dàng kế thừa, mở rộng
* Thành phần của Iterator pattern do các interface/abstract class quy định
* Đôi khi dùng interator có thể kém hiệu quả hơn cách duyệt thông thường tùy vào cách cài đặt

### Ví dụ thực tế

Như đã nói ở trên, thực thế hầu hết các ngôn ngữ lập trình đều đã implement Iterator để phục vụ cho việc indexing. Ví dụ:

* C++: <https://www.cplusplus.com/reference/iterator/>
* C#: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators>
* Java: <https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>

## Mediator

### Tổng quan

* Tên: Mediator.
* Phân loại: Mediator thuộc nhóm Behavioral Pattern.
* Mục đích, ý định: giúp giảm thiểu sự phụ thuộc phức tạp giữa các object với nhau bằng cách giao tiếp thông qua 1 object trung gian (**mediator**).

### Khả năng ứng dụng

* Khi mức độ phụ thuộc của các component phức tạp nhưng vẫn muốn đảm bảo khả năng tái sử dụng của component.
* Khi các component chỉ phụ thuộc nhau trong 1 ngữ cảnh nhất định.

### Cấu trúc

Thiết kế chỉ có duy nhất 1 thành phần là **Mediator**, trong đó:

* **Mediator**: là 1 Interface và hàm **notify**. Hàm này có tác dụng gửi thông điệp từ 1 component đến 1 component khác.
* **ConcreteMediator**. Implement Mediator và giữ tham chiếu đến các component trong hệ thống.

Các component giữ tham chiếu đến interface **Mediator**. Khi muốn gửi đi thông điệp hoặc thông tin sự kiện, component sẽ gọi hàm **notify** của mediator.

### Mã nguồn minh họa

Xây dựng phòng chat cho phép người dùng gửi và nhận tin nhắn của nhau. Phòng chat sẽ có các sự kiện thông báo như: người dùng mới tham gia phòng, người dùng gửi tin nhắn, người dùng rời phòng.

Source code: <https://github.com/LichND/Design-Pattern/tree/main/mediator>

### Hệ quả

* Đảm bảo nguyên tắc Single Responsibility: Lúc này các component không cần phải thêm các chức năng giúp giao tiếp với các component khác. Từ đó tăng khả năng tái sử dụng component.
* Đảm bảo nguyên tắc Open/Closed.
* Đảm bảo nguyên tắc Dependency Inversion: Lúc này thay vì các component phụ thuộc lẫn nhau thì luồng phụ thuộc được đẩy sang 1 lớp có mức cao hơn là **Mediator**.

*Nhược điểm:*

* Khi Mediator trở nên quá phức tạp thì sẽ rất khó quản lí.
* Chỉ phát huy tối đa công dụng khi các sự kiện được thông báo ở mức global.

### Ví dụ thực tế

**Phòng chat**

**Form**

Khi xây dựng các form, các component trong form có thể phụ thuộc lẫn nhau (như khi ta chọn check box hiện mật khẩu thì textbox mật khẩu sẽ chuyển sang chế độ text hay vì mask). Lúc này, ta sử dụng Form như 1 Mediator, các component khác sẽ giao tiếp với nhau thông qua Form.

### Các mẫu liên quan

**Observer**

Cả 2 mẫu đều nhằm mục đích truyền đạt thông tin đến các thành phần của hệ thống.**Observer**

* Các subscriber nhận thông điệp từ 1 nguồn duy nhất (ở đây là observer).
* Event-based.

**Mediator**

* Các component có thể nhận thông điệp từ nhiều nguồn khác nhau (các component khác).
* - Object-based.**Façade**

Tổ chức lại các thành phần để giảm bớt các sự phụ thuộc không cần thiết. Tuy nhiên, nếu như Façade cung cấp một interface đại diện cho thành phần hệ thống để giao tiếp với các hệ thống khác thì Mediator chuyển đổi sự phụ thuộc của các component về 1 lớp trung tâm.

## Memento

### Tổng quan

* Tên: Memento.
* Phân loại: Memento thuộc nhóm Behavioral Pattern.
* Mục đích, ý định: giúp lưu trữ và khôi phục trạng thái của 1 object mà không tiết lộ quá nhiều về chi tiết của object đó.

### Khả năng ứng dụng

* Khi cần quản lí (lưu trữ, truy xuất) toàn bộ thông tin state của 1 object.
* Có thể sử dụng trong transaction để rollback khi có lỗi.

### Cấu trúc

*Mục tiêu thiết kế:*

* Không để public state ở cả component lẫn Memento
* Chỉ có component mới tác động trực tiếp đến state của Memento.

Graphical user interface, application

Description automatically generated**Giải pháp 1**: Sử dụng nested class

Gồm 3 thành phần:

**Component**: thành phần cần quản lí state. Có 2 hàm:

* save: lưu trữ lại state của component. Hàm này trả về Memento.
* restore: Khôi phục lại state của component. Tham số nhận vào là 1 memento.

**Memento**: Lưu trữ state của component. Đây là lớp nằm bên trong lớp Component. Do đó chỉ có Component có thể access trực tiếp các thành phần bên trong Memento.

**ComponentWrapper**: có tham chiếu đến component và danh sách các Memento của component cần quản lí. Có thể hiểu ComponentWrapper như 1 component nhưng được cài đặt thêm chức năng quản lí state.

*Nhược điểm*: Nested class chỉ có ở một số ngôn ngữ nhất định.

**Giải pháp 2:** Sử dụng interface trung gianGraphical user interface

Description automatically generated

Thiết kế này cũng gồm 3 thành phần như trong giải pháp 1. Tuy nhiên lúc này Memento bao gồm 1 interface rỗng và class **ConcreteMemento** quản lí state của component. ComponentWrapper quản lí trạng thái của component thông qua interface **Memento**.

Đây là giải pháp thay thế cho giải pháp 1 đối với những ngôn ngữ không hỗ trợ nested class.

Graphical user interface, application

Description automatically generated**Giải pháp 3:** Sử dụng Memento như một wrapper với component.

Khác với 2 giải pháp đầu, giải pháp 3 sử dụng Memento như 1 wrapper của component. **ConcreteMemento** lúc này sẽ giữ tham chiếu đến component.

Interface **Memento** có hàm **restore**. Khi hàm này được gọi, component của Memento sẽ được gán lại state của Memento hiện tại.

**StateManager**: quản lí các Memento của component. StateManager có thể quản lí nhiều memento của nhiều component khác nhau.

### Mã nguồn minh họa

Source code: <https://github.com/LichND/Design-Pattern/tree/main/Mementor>

### Hệ quả

* Đảm bảo nguyên tắc **Single Responsibility**: lúc này việc quản lí history của state do 1 thành phần khác đảm nhiệm.
* Đảm bảo nguyên tắc **Open/Closed**: cung cấp 1 access modifier phù hợp, không để lộ các thông tin quan trọng của component

*Nhược điểm:*

* Tốn bộ nhớ nếu lưu trữ quá nhiều thông tin trong memento hoặc quản lí quá nhiều memento.
* StateManager cần phải theo dõi trong trường hợp component không còn tồn tại. Trong trường hợp vòng đời của component đã kết thúc thì StateManager phải có giải pháp để loại bỏ memento tương ứng của component đó.

### Các mẫu liên quan

**Command**

A picture containing shape

Description automatically generatedKết hợp với Command để hiện thực hóa chức năng undo.

## Observer

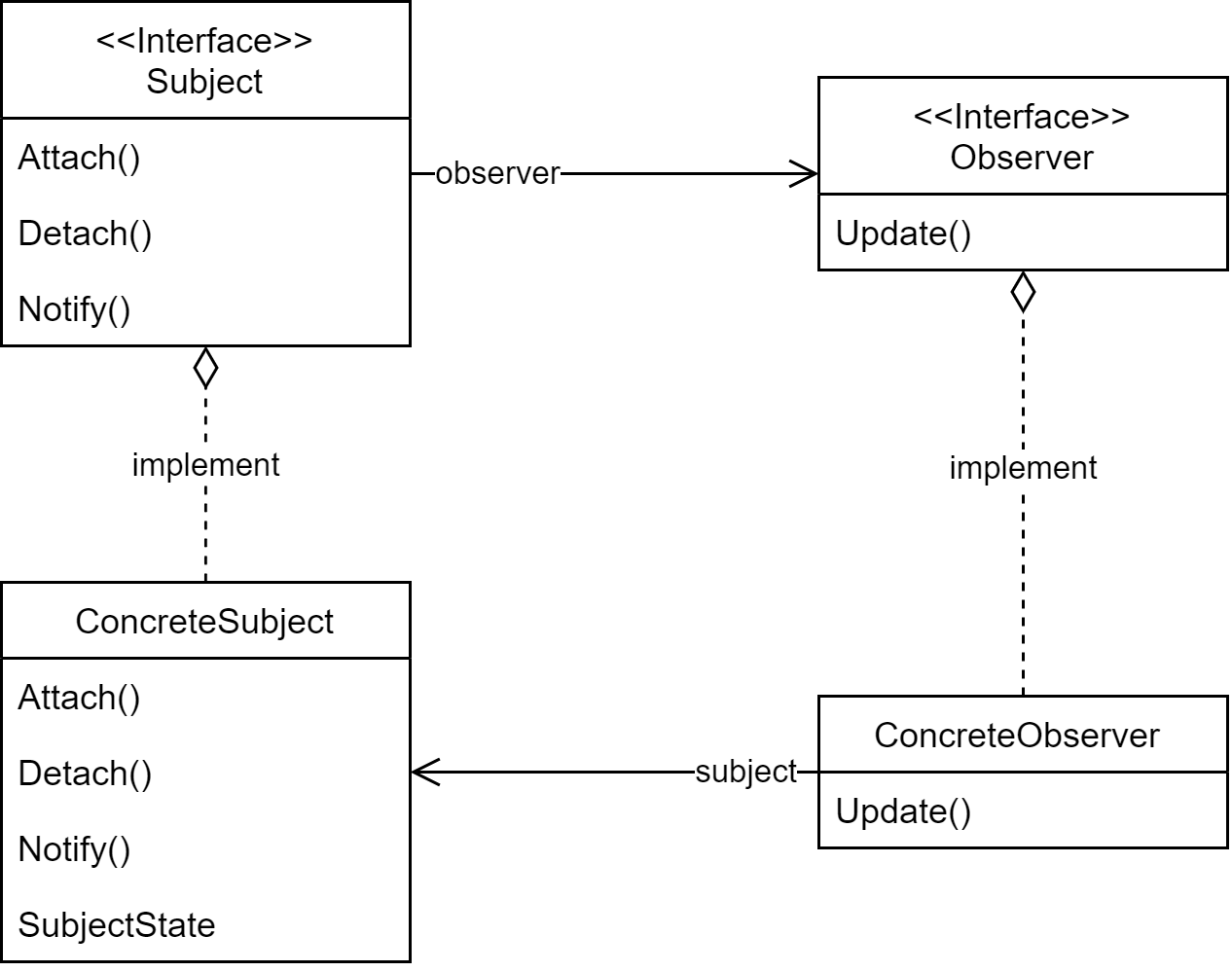
### Tổng quan

* Tên: Observer.
* Phân loại: Observer thuộc nhóm Behavior Pattern.
* Bí danh: Observateur.
* Mục đích, ý định: Định nghĩa sự phụ thuộc một-nhiều giữa các đối trượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.

### Khả năng ứng dụng

* Khi một Observer đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ báo cho toàn bộ Observer đã đăng ký biết. Khi không cần nữa thì Observer này sẽ được gỡ khỏi hệ thống

### Cấu trúc



Các thành phần tham gia Observer Pattern:

* **Subject**: chứa danh sách các observer, cung cấp phương thức để có thể thêm và loại bỏ observer.
* **Observer**: định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
* **ConcreteSubject**: cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
* **ConcreteObserver**: cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

Sự tương tác giữa subject và các observer như sau: mỗi khi subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các observer của nó và gọi phương thức cập nhật trạng thái ở từng observer, có thể truyền thêm thông tin để các observer có thể lấy ra trạng thái của nó và xử lý.

### Mã nguồn minh họa

Observer được dùng chủ yếu như một hệ thống quản lý event mà đây là một ví dụ đơn giản: <https://github.com/LichND/Design-Pattern/tree/main/Observer>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc
* Các class con dễ dàng thay đổi class cha mà không làm ảnh hưởng đến chương trình
* Các interface có thể dễ dàng kế thừa, mở rộng
* Thành phần của Observer pattern do các interface/abstract class quy định

### Ví dụ thực tế

Observer cũng được nhiều ngôn ngữ implement sẵn:

* C#: <https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern>
* Java: <https://docs.oracle.com/javase/7/docs/api/java/util/Observer.html>
* JavaScript: <https://developer.mozilla.org/en-US/docs/Archive/Web/JavaScript/Object.observe>

## State

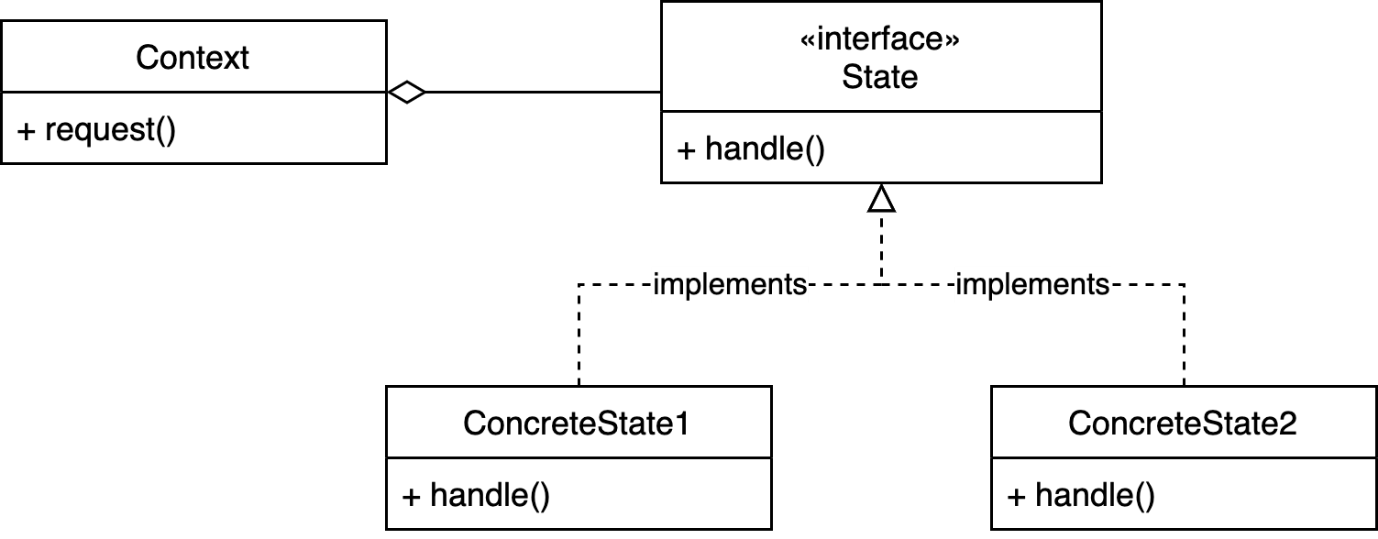
### Tổng quan

* Tên: State.
* Phân loại: State thuộc nhóm Behavior Pattern.
* Bí danh: Etat.
* Mục đích, ý định: Cho phép thay đổi ứng xử của đối tượng tùy theo sự thay đổi trạng thái bên trong của nó

### Khả năng ứng dụng

* Khi hành vi của đối tượng phụ thuộc vào trạng thái của nó và nó phải có khả năng thay đổi hành vi của nó lúc run-time theo trạng thái mới.
* Khi nhiều điều kiện phức tạp buộc đối tượng phụ thuộc vào trạng thái của nó.
* Khi yêu cầu của việc thêm trạng thái cho đối tượng là dễ dàng.

### Cấu trúc



Các thành phần tham gia State Pattern:

* **Context**: được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
* **State**: là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
* **ConcreteState**: cài đặt các phương thức của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

### Mã nguồn minh họa

**Vấn đề**: Cho một trò chơi với Player có các hành vi sau:

* Idle
* Run
* Die

Mỗi hành vi đều có hàm update và render riêng. Ngoài ra các state này đều phải tự động được thay đổi dựa vào nội tại của Player. Viết chương trình mô phỏng

<https://github.com/LichND/Design-Pattern/tree/main/State>

### Hệ quả

* Dễ dàng thay đổi sửa chữa nhưng vẫn đảm bảo sự ổn định của interface gốc.
* Các class con dễ dàng thay đổi class cha mà không làm ảnh hưởng đến chương trình.
* Các interface của State có thể dễ dàng kế thừa, mở rộng.
* Thành phần của  State do interface/abstract class quy định.
* Giúp chuyển trạng thái một cách rõ ràng

### Ví dụ thực tế

* 100% các game hiện nay đều có sự góp mặt của state pattern
* 100% các ứng dụng quản lý cũng có sự góp mặt của state pattern

## Strategy

### Tổng quan

* Tên: Strategy,
* Phân loại: Abstract Factory thuộc nhóm Behavioral Pattern.
* Mục đích, ý định: Tách các “**chiến lược**” cụ thể cho một hành vi nào đó thành các class khác nhau. Class đảm nhiệm vai trò chọn “chiến lược” và thực thi được gọi là **context.**

### Khả năng ứng dụng

* Khi có nhiều biến thể về cách giải quyết cho cùng 1 vấn đề và các biến thể này có thể thay thế lẫn nhau.
* Sử dụng khi cần tách việc cài đặt ra khỏi ngữ cảnh của class hiện tại (nhất là khi chi tiết việc cài đặt không quá quan trọng).
* Giải pháp hiệu quả để thay thế các toán tử điều kiện phức tạp.

Nhược điểm:

* Các chiến lược chịu ràng buộc bởi interface.
* Đôi khi sử dụng strategy sẽ làm phức tạp hóa vấn đề.

### Cấu trúc

Bao gồm 2 thành phần chính:

**Strategy:**  Bao gồm các thành phần quy định chiến lược thực thi. Cụ thể ở đây là 1 interface **Strategy** và 2 class **ConcreteStrategyA** và **ConcreteStrategyB**.

**Context:** Ngữ cảnh sử dụng của strategy. Context có giữ tham chiếu đến interface **Strategy**. Context có vai trò lựa chọn Strategy để thực thi ở client bằng hàm setStrategy.

### Mã nguồn minh họa

* Thiết kế 1 module để xác thực thông tin người dùng (authentication).
* Có 3 phương thức xác thực:
  + Xác thực bằng tài khoản và mật khẩu.
  + Xác thực bằng token
  + Xác thực bằng bên thứ 3.

Source code: <https://github.com/LichND/Design-Pattern/tree/main/Strategy>

### Hệ quả

* Thể hiện rõ ràng nhất ý tưởng “Kết hợp hơn là kế thừa” (composition over inheritance). Vì kế thừa không phải là cách duy nhất để tái sử dụng code. Ngoài ra kế thừa còn có hạn chế đó là ràng class con phụ thuộc vào class cha, trong khi composition thì không.
* Abstraction over implementation.
* Tuân thủ nguyên tắc mở/đóng (**Open-Closed principle**).
* Tuân thủ nguyên tắc **Liskov substitution.**

### Ví dụ thực tế

**Passport**

Passport là 1 package được sử dụng khá phổ biến trong NodeJS. Cụ thể, passport hỗ trợ nhiều loại xác thực khác nhau bằng cách tách các loại xác thực thành nhiều Strategy khác nhau, qua đó giúp tối ưu hóa việc tái sử dụng.

### Các mẫu liên quan

## Template Method

### Tổng quan:

- Tên: Template Method.

- Phân loại: Template Method thuộc nhóm Behavior.

- Bí danh: Patron de méthode.

- Mức độ phổ biến: Trung bình.

- Mục đích, ý định: Định nghĩa ra phần khung của một thuật toán nào đó, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc hiện thực các phương thức được ủy nhiệm cho các lớp con.

### Khả năng ứng dụng:

- Sử dụng mẫu Template Method khi muốn cho phép người dùng chỉ mở rộng các bước cụ thể của một thuật toán, nhưng không mở rộng cấu trúc của nó.

- Sử dụng mẫu khi bạn có một số lớp chứa các thuật toán gần như giống hệt nhau với một số khác biệt nhỏ.

### Cấu trúc:



***Source:*** [***https://refactoring.guru/design-patterns***](https://refactoring.guru/design-patterns)

***-* Abstract Class** khia báo các phương thức thực hiện các bước của một thuật toán, Một bước của giải thuật có thể implement từ Abstract Class hoặc có một implemtation mặc định.

- **Concrete Classes** có thể override lại các bước của giải thuật, nhưng không đổi được bản thân thuật toán.

### Mã nguồn minh họa:

- Trong ví dụ bên dưới, minh hoạ các bước để trình bày một buổi Seminar sử dụng silde theo một trình tự nhất định. Khi Seminar các nội dung khác nhau thì người dùng chỉ cần thay đổi nội dung từng giai đoạn Seminar chứ không thay đổi được trình tự các nội dung.

- https://github.com/LichND/Design-Pattern/tree/main/TemplateMethod

### Hệ quả:

\*Ưu điểm:

- Cho phép người dùng override lại các phần cụ thể của một giải thuật lớn và hạn chế tác động của nó tới các phần khác.

- Các code giống nhau có thể cho vào một super class.

\*Nhược điểm:

- Các client bị giới hạn bởi việc cung cấp một bộ khung cho sẵn.

- Có thể vi phạm nguyên lý Liskov do override lại các bước mặc định trong các class con.

- Nếu như giải thuật có quá nhiều bước, thì template method sẽ khó bảo trì hơn.

### Ví dụ thực tế:

- Một số thư viện Java sử dụng Template method: tất cả các non-abstract class của các thư viện: java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer, java.util.AbstractList, java.util.AbstractSet và java.util.AbstractMap.

### Các mẫu liên quan:

- Factory Method là một specialization của Template Method. Đồng thời Factory Method có thể là một bước trong Template Method.

## Visitor

### Tổng quan

* Tên: Visitor.
* Phân loại: Visitor thuộc nhóm Behavioral Pattern.
* Mục đích, ý định: gồm 2 thành phần:
* Visitor: là thành phần cài đặt các hàm mà **Element** cần gọi.
* Element: Thành phần của hệ thống có vai trò lựa chọn hàm cần gọi ở **Visitor.**

### Khả năng ứng dụng

* Khi có quá nhiều loại đối tượng cần mở rộng thêm cùng 1 chức năng mới.
* Tách các logic không liên quan tới thành phần chính và cài đặt chúng trong một thành phần khác.
* Nếu chức năng mới cần cài đặt chỉ tương thích với một số thành phần nhất định.

### Cấu trúc

Gồm 2 thành phần:

**Element**: Bao gồm các interface và class của hệ thống đã có từ trước. Mỗi class Element có hàm **accept**. Hàm này nhân vào tham số là 1 interface **Visitor**.

**Visitor:** Thành phần chứa các hàm để các đối tượng trong Element có thể gọi được. Mỗi hàm **visit** như trong hình sẽ được cài đặt tương ứng cho mỗi class bên trong Element.

### Mã nguồn minh họa

Đặt vấn đề:

* Một bệnh viện cần quản lí thông tin của nhân viên. Có 3 loại nhân viên: Bác sĩ, y tá, bảo vệ.
* Thông tin quản lí bao gồm các thông tin cơ bản như: họ tên, ngày sinh, giới tính.
  + Bác sĩ: trình độ học vấn.
  + Bảo vệ: công ty đã thuê.

**Mở rông yêu cầu:**

Hệ thống cần tính và tổng hợp lương của các nhân viên. Trong đó, mức lương được tính như sau:

* + Bác sĩ: lương = 1.5 x lương cơ bản
  + Y tá: lương = số ngày làm việc x 300,000.
  + Bảo vệ: công ty đã thuê sẽ chịu trách nhiệm chi trả.

Soure code: <https://github.com/LichND/Design-Pattern/tree/main/Visitor>

### Hệ quả

* Tuân thủ nguyên tắc đóng / mở (Open/Closed Principle).
* Tuân thủ nguyên tắc Single Responsibility. Đảm bảo Element chỉ tập trung vào chức năng chính của nó. Việc mở rộng các hành vi được thực thi bên trong Visitor.
* Có thể hoạt động hiệu quả trên các cấu trúc phức tạp của Element (cây, đồ thị, … ).

Nhược điểm:

* Khi có 1 element thay đổi thì tất cả Visitor có cài đặt của element đó cũng sẽ có khả năng bị thay đổi.
* Visitor không có quyền truy cập hoàn toàn tới Element.
* Cần public một số thông tin từ Element.
* Vi phạm quy tắc đóng gói.

### Các mẫu liên quan

**Composite**: có thể sử dụng Visitor để mở rộng và thực thi hành vi cho các thành phần của Composite.

# Tài liệu tham khảo

C++ reference: <https://www.cplusplus.com/reference/>

C# document: <https://docs.microsoft.com/en-us/dotnet/csharp/>

Java document: <https://docs.oracle.com/en/java/>

JavaScript document: <https://developer.mozilla.org/en-US/docs/Web/API/Document>

Refactoring.guru: <https://refactoring.guru/design-patterns>

Wikipedia: <https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern>

Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike, biên tập. “Head First Design Patterns”

Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software.

Alexander, Shvets (2019). Dive Into Design Patterns

…

**NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN**

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

\_\_\_\_\_\_*\_\_\_\_\_\_ , ngày \_\_\_ tháng \_\_\_ năm 2021*

***Người nhận xét***