

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN MÔN
MẪU THIẾT KẾ**

**TÌM HIỂU MẪU THIẾT KẾ
CHAIN OF RESPONSIBILITY
VÀ INTERPRETER**

Người hướng dẫn: **THẦY Vũ Đình Hồng**

Người thực hiện: **PHẠM THÀNH PHƯƠNG – 51603247**

Khoá : 20

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN MÔN
MẪU THIẾT KẾ**

**TÌM HIỂU MẪU THIẾT KẾ
CHAIN OF RESPONSIBILITY
VÀ INTERPRETER**

Người hướng dẫn: **THẦY Vũ Đình Hồng**

Người thực hiện: **PHẠM THÀNH PHƯƠNG – 51603247**

Khoá : 20

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

LỜI CẢM ƠN

Đề bài tập lớn này được hoàn thành đúng tiến độ và đạt kết quả tốt đẹp, ngoài sự nỗ lực của em, còn có sự giúp đỡ, đóng góp ý kiến và hướng dẫn nhiệt tình của thầy cô trong khoa Công nghệ thông tin giúp em có thêm động lực để hoàn thành đồ án này.

Em xin gửi lời cảm ơn chân thành đến thầy Vũ Đình Hồng đã tận tình hướng dẫn em vượt qua khó khăn, vướng mắc trong suốt thời gian làm bài tập lớn.

Em cũng xin chân thành cảm ơn toàn thể giảng viên, công nhân viên chức trường Đại học Tôn Đức Thắng đã tạo điều kiện, môi trường giáo dục tốt đẹp, cung cấp tài liệu và kiến thức đầy đủ giúp em có cơ sở lý thuyết vững vàng và tạo điều kiện giúp đỡ em trong quá trình học tập.

Với trình độ chuyên môn còn hạn chế, đồ án có bị thiếu sót và sai lầm ở một số chỗ mà em không tìm ra. Em rất mong nhận được sự chỉ dẫn, đóng góp ý kiến của thầy và bạn bè để em có thể bổ sung, cập nhật những sai lầm sau khi đồ án kết thúc, qua đó nâng cao ý thức của em, phục vụ tốt cho các dự án thực tế sau này.

Em xin chân thành cảm ơn.

Tp. HCM, ngày 7 tháng 05 năm 2021

Tác giả

(Ký và ghi rõ họ tên)

Phạm Thành Phương

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng em xin cam đoan bài tập lớn “**Mẫu thiết kế**” là công trình nghiên cứu của sinh viên Phạm Thành Phương (51603247). Những phần sử dụng tài liệu tham khảo trong đồ án đã được nêu rõ trong phần tài liệu tham khảo. Các số liệu, kết quả trình bày trong đồ án là hoàn toàn trung thực, nếu sai phạm em xin chịu hoàn toàn trách nhiệm và chịu mọi kỷ luật của bộ môn và nhà trường.

Nếu phát hiện có bất kỳ sự gian lận nào em xin hoàn toàn chịu trách nhiệm về nội dung đồ án của em. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 07 tháng 05 năm 2021

Tác giả

(ký tên và ghi rõ họ tên)

Phạm Thành Phương

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Phần đánh giá của GV chấm bài

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

TÓM TẮT

Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, nó khá quan trọng và mọi lập trình viên muốn giỏi đều phải biết. Được sử dụng thường xuyên trong các ngôn ngữ OOP. Nó sẽ cung cấp cho bạn các "mẫu thiết kế", giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

Design Patterns không phải là ngôn ngữ cụ thể nào cả. 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.

Trong bài tập lớn này sẽ đi sâu hơn và tìm hiểu về hai pattern cụ thể là Chain of Responsibility và Interpreter. Mỗi mẫu thiết kế sẽ được áp dụng và sử dụng cho mỗi mục đích khác nhau và chúng đều rất hữu ích khi thiết kế một chương trình.

MỤC LỤC

| | |
|---|----|
| LỜI CẢM ƠN | 4 |
| PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN | 6 |
| TÓM TẮT | 7 |
| MỤC LỤC | 8 |
| CHƯƠNG 1 – CHAIN OF RESPONSIBILITY PATTERN | 9 |
| 1.1 Đặt vấn đề | 9 |
| 1.2 Giới thiệu về Chain Of Responsibility Pattern | 9 |
| 1.2.1 Ví dụ thực tiễn | 9 |
| 1.2.3 Chain of Responsibility | 9 |
| 1.2.4 Lợi ích khi sử dụng | 10 |
| 1.2.5 Nên sử dụng khi nào | 10 |
| 1.2.6 Cài đặt Chain of Responsibility pattern như thế nào | 11 |
| 1.3 Ví dụ | 12 |
| 1.4 Áp dụng | 12 |
| 1.5 Kết luận | 17 |
| CHƯƠNG 2 – INTERPRETER PATTERN | 18 |
| 2.1 Đặt vấn đề | 18 |
| 2.2 Giới thiệu về Interpreter Pattern | 18 |
| 2.2.1 Interpreter Pattern là gì | 18 |
| 2.2.2 Cài đặt Interpreter Pattern | 19 |
| 2.2.3 Lợi ích của Interpreter Pattern | 20 |
| 2.2.4 Sử dụng Interpreter Pattern khi nào | 20 |
| 2.3 Ví dụ | 21 |
| 2.4 Áp dụng | 23 |
| 2.5 Kết luận | 28 |
| TÀI LIỆU THAM KHẢO | 29 |

CHƯƠNG 1 – CHAIN OF RESPONSIBILITY PATTERN

1.1 Đặt vấn đề

Trong một ứng dụng được xây dựng sẽ luôn có lỗi xảy ra. Điều này là chắc chắn, nhưng tùy thuộc vào loại lỗi và cấp độ nguy hiểm của từng lỗi mà chúng ta cần tạo ra các cảnh báo khác nhau. Ví dụ lỗi thông thường chỉ cần in ra cmd là có thông báo có lỗi, có lỗi lớn hơn thì ghi ra file log để ghi chép lại, và nếu lỗi đó cực lớn thì cần email trực tiếp cho người có khả năng xử lý lỗi đó ngay lập tức.

Từ bên trên chúng ta có thể thấy rằng vấn đề trên cần được giải quyết tùy thuộc vào mức độ, tính nghiêm trọng của sự việc mà đưa ra những quyết đoán hợp lý.

1.2 Giới thiệu về Chain Of Responsibility Pattern

1.2.1 Ví dụ thực tiễn

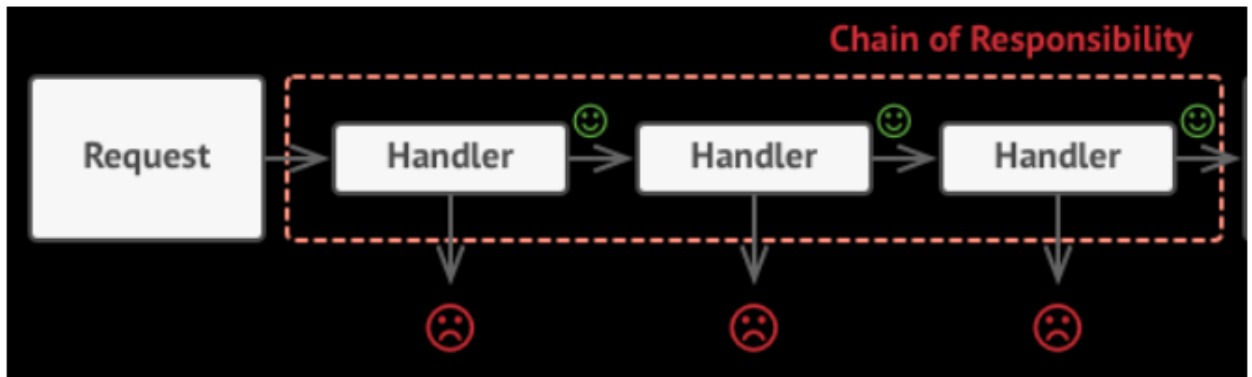
Giờ chúng ta hãy lấy ví dụ thực tiễn trong cuộc sống để minh họa về Chain Of Responsibility Pattern :

Hãy tưởng tượng bạn đi đến ủy ban phường để làm một thủ tục hành chính nào đó. Bạn tiếp xúc với người phụ trách ở bộ phận một cửa và đưa ra yêu cầu công việc. Với những công việc đơn giản (như đóng dấu công chứng), người đó có thể thực hiện ngay cho bạn. Nhưng nếu yêu cầu phức tạp hơn mà người phụ trách phòng một cửa không trực tiếp làm được, người đó sẽ chuyển yêu cầu đó đến người phụ trách mảng công việc tương ứng. Nếu yêu cầu công việc có gì đó bất thường mà người phụ trách chính mảng công việc đó cũng không xử lý được, người đó sẽ chuyển tiếp đơn lên chủ tịch/phó chủ tịch phường để quyết định cách xử lý. Tương tự, nếu ở cấp độ phường không xử lý được, yêu cầu sẽ chuyển tiếp lên cấp quận/huyện.

1.2.3 Chain of Responsibility

Chain of Responsibility (COR) là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Chain of Responsibility cũng tương tự như cơ chế bên trên, kết nối người gửi một yêu cầu đến nơi nhận yêu cầu của nó bằng cách cho nhiều hơn một đối tượng một cơ hội để xử lý các yêu cầu. Chuỗi các đối tượng tiếp nhận và truyền các yêu cầu theo chuỗi cho đến khi một đối tượng tiếp nhận xử lý nó. Khởi tạo và chạy lại yêu cầu với một

đường ống xử lý duy nhất có chứa nhiều xử lý khả thi. Trong mô hình này, thông thường mỗi đối tượng tiếp nhận yêu cầu có chứa tham chiếu đến đối tượng tiếp nhận yêu cầu khác. Nếu một đối tượng không thể xử lý các yêu cầu, nó sẽ gửi yêu cầu đó thông qua message đến đối tượng tiếp theo.



1.2.4 Lợi ích khi sử dụng

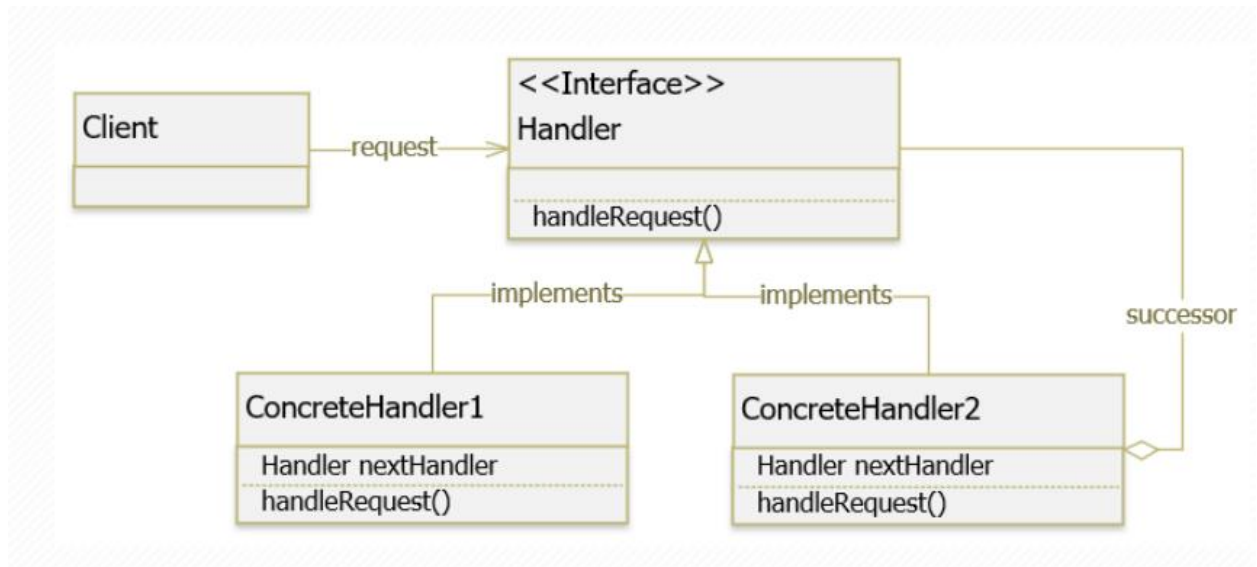
- Giảm kết nối (loose coupling): Thay vì một đối tượng có khả năng xử lý yêu cầu chứa tham chiếu đến tất cả các đối tượng khác, nó chỉ cần một tham chiếu đến đối tượng tiếp theo. Tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu (sender) và các đối tượng nhận yêu cầu (receivers).
- Tăng tính linh hoạt : đảm bảo Open/Closed Principle.
- Phân chia trách nhiệm cho các đối tượng: đảm bảo Single Responsibility Principle.
- Có khả năng thay đổi dây chuyền (chain) trong thời gian chạy.
- Không đảm bảo có đối tượng xử lý yêu cầu.

1.2.5 Nên sử dụng khi nào

- Có nhiều hơn một đối tượng có khả năng xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
- Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
- Khi cần phải thực thi các trình xử lý theo một thứ tự nhất định..

- Khi một tập hợp các đối tượng xử lý có thể thay đổi động: tập hợp các đối tượng có khả năng xử lý yêu cầu có thể không biết trước, có thể thêm bớt hay thay đổi thứ tự sau này.

1.2.6 Cài đặt Chain of Responsibility pattern như thế nào



Các thành phần tham gia mẫu Chain of Responsibility:

Handler : định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).

ConcreteHandler : xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreteHandler không thể xử lý được yêu cầu, nó sẽ gọi lại yêu cầu cho successor của nó.

Client : tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.

Client gửi một yêu cầu để được xử lý gửi nó đến chuỗi (chain) các trình xử lý (handlers), đó là các lớp mở rộng lớp Handler. Mỗi Handler trong chuỗi lần lượt cố gắng xử lý yêu cầu nhận được từ Client. Nếu trình xử lý đầu tiên (ConcreteHandler) có thể xử lý nó, thì yêu cầu sẽ được xử lý. Nếu không được xử lý thì sẽ gửi đến trình xử lý tiếp theo trong chuỗi (ConcreteHandler + 1).

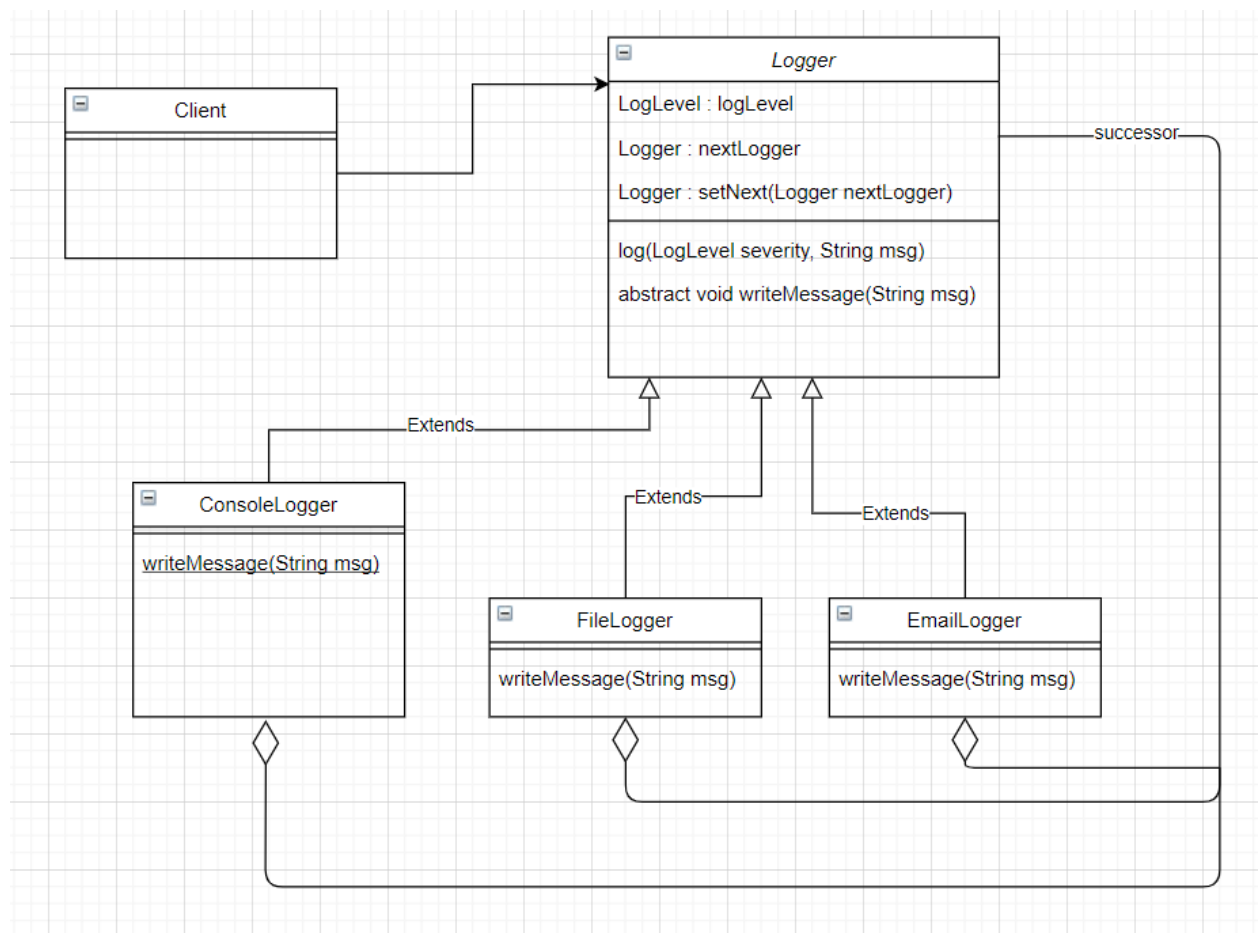
1.3 Ví dụ

Chúng ta sẽ lấy luôn vấn đề cài đặt Logger cho một ứng dụng làm ví dụ cho lần này. Lớp Logger sẽ chịu trách nhiệm ghi log ra trong trường hợp bị lỗi hoặc có vấn đề cần phải ghi chép lại. Tùy từng tình huống, mức độ nghiêm trọng của sự việc mà chúng ta có các mức độ thực thi khác nhau. Các mức độ log là : None, Infor – ghi thông tin, Debug – debug, Warning – Cảnh báo, Error – lỗi, Fatal – Cực nghiêm trọng, All – thực hiện tất cả.

1.4 Áp dụng

Sơ đồ class được vẽ trên <https://app.diagrams.net/> lưu tại file : documents/Bai tap lon - Mau thiet ke.drawio

Sơ đồ class :




Đầu tiên chúng ta tạo ra enum (constant) LogLevel sử dụng để xác định mức độ chúng ta cần ghi log. Enum trong java có thể thiết lập Constructor, link tham khảo ví dụ về enum constructor <https://www.programiz.com/java-programming/enum-constructor> .


```
1  public enum LogLevel {
2      NONE(0), INFO(1), DEBUG(2), WARNING(4), ERROR(8), FATAL(16), ALL(32);
3
4      private int level;
5
6      private LogLevel(int level) {
7          this.level = level;
8      }
9
10     public int getLevel() {
11         return level;
12     }
13 }
```

Tạo abstract class `Logger` : cho phép thực hiện một chain logger dựa vào giá trị `LogLevel` ứng với từng Handler. Nếu mức độ lỗi (severity) lớn hơn hoặc bằng với `LogLevel` mà nó có thể handle thì sẽ thực hiện `writeMessage()`, đồng thời gọi Handler kế tiếp nếu có.

```
1  public abstract class Logger {
2
3      protected LogLevel logLevel;
4
5      protected Logger nextlogger; // The next Handler in the chain
6
7      public Logger(LogLevel logLevel) {
8          this.logLevel = logLevel;
9      }
10
11     // Set the next logger to make a list/chain of Handlers.
12     public Logger setNext(Logger nextlogger) {
13         this.nextlogger = nextlogger;
14         return nextlogger;
15     }
16
17     public void log(LogLevel severity, String msg) {
18         if (logLevel.getLevel() <= severity.getLevel()) {
19             writeMessage(msg);
20         }
21         if (nextlogger != null) {
22             nextlogger.log(severity, msg);
23         }
24     }
25
26     protected abstract void writeMessage(String msg);
27 }
```

ConsoleLogger, FileLogger, EmailLogger thừa kế abstract class Logger

```
1 public class ConsoleLogger extends Logger {  
2       
3     public ConsoleLogger(LogLevel logLevel) {  
4         super(logLevel);  
5     }  
6  
7     @Override  
8     protected void writeMessage(String msg) {  
9         System.out.println("Console logger: " + msg);  
10    }  
11 }
```

```
1 public class FileLogger extends Logger {  
2       
3     public FileLogger(LogLevel logLevel) {  
4         super(logLevel);  
5     }  
6  
7     @Override  
8     protected void writeMessage(String msg) {  
9         System.out.println("File logger: " + msg);  
10    }  
11 }
```

```
1 public class EmailLogger extends Logger {  
2  
3     public EmailLogger(LogLevel logLevel) {  
4         super(logLevel);  
5     }  
6  
7     @Override  
8     protected void writeMessage(String msg) {  
9         System.out.println("Email logger: " + msg);  
10    }  
11 }
```

Mức độ từng Logger cụ thể xử lý :

- NONE, INFO, DEBUG, WARNING : ConsoleLogger
- ERROR : ConsoleLogger, FileLogger
- FATAL, ALL : ConsoleLogger, FileLogger, EmailLogger

```
1 public class AppLogger {
2
3     public static Logger getLogger() {
4         Logger consoleLogger = new ConsoleLogger(LogLevel.DEBUG);
5         Logger fileLogger = consoleLogger.setNext(new FileLogger(LogLevel.ERROR));
6         fileLogger.setNext(new EmailLogger(LogLevel.FATAL));
7         return consoleLogger;
8     }
9 }
```

Và cuối cùng là thiết lập Client để gọi đến Logger

```
2 public class Client {
3
4     Run | Debug
5     public static void main(String[] args) {
6         // * Tạo chain of responsibility
7         Logger logger = AppLogger.getLogger();
8
9         // * thử log ở mức độ Debug
10        System.out.println("Log level Console");
11        //logger.log(LogLevel.INFO, "Info message");
12        logger.log(LogLevel.DEBUG, "Debug message");
13        //logger.log(LogLevel.WARNING, "Debug message");
14
15        // * thử log ở mức độ Error
16        System.out.println("----\nLog level Error - loi");
17        logger.log(LogLevel.ERROR, "Error message");
18
19        // * thử log ở mức độ Fatal
20        System.out.println("----\nLog level Fatal - cuc nguy hiem");
21        logger.log(LogLevel.FATAL, "Fatal message");
22    }
23 }
```


Toàn bộ code được lưu tại : source\Chain of Responsibility Pattern

Cách chạy :

- 1 javac *.java
- 2 java Client

Kết quả :

```
C:\Hoctap\MauThietKe\doan\Mau-thiet-ke-doan\Baitaplon\source\Chain of Responsibility Pattern>java Client
Log level Console
Console logger: Debug message
----
Log level Error - loi
Console logger: Error message
File logger: Error message
----
Log level Fatal - cuc nguy hiem
Console logger: Factal message
File logger: Factal message
Email logger: Factal message
```

1.5 Kết luận

Chain of Responsibility (COR) là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Chain of Responsiblity cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

CHƯƠNG 2 – INTERPRETER PATTERN

2.1 Đặt vấn đề

Để giao tiếp hoặc cho người dùng hiểu chương trình, ứng dụng chúng ta cần hiểu thị ngôn ngữ của người dùng đó. Như ứng dụng gọi xe grab, để phổ biến số lượng người có thể sử dụng ứng dụng chúng ta cần sử dụng ngôn ngữ là tiếng việt. Nên khi thiết lập ứng dụng chúng ta cần một bộ chuyển đổi từ tiếng anh sang tiếng việt để người dùng có thể dễ dàng sử dụng ứng dụng

2.2 Giới thiệu về Interpreter Pattern

2.2.1 *Interpreter Pattern là gì*

Interpreter Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

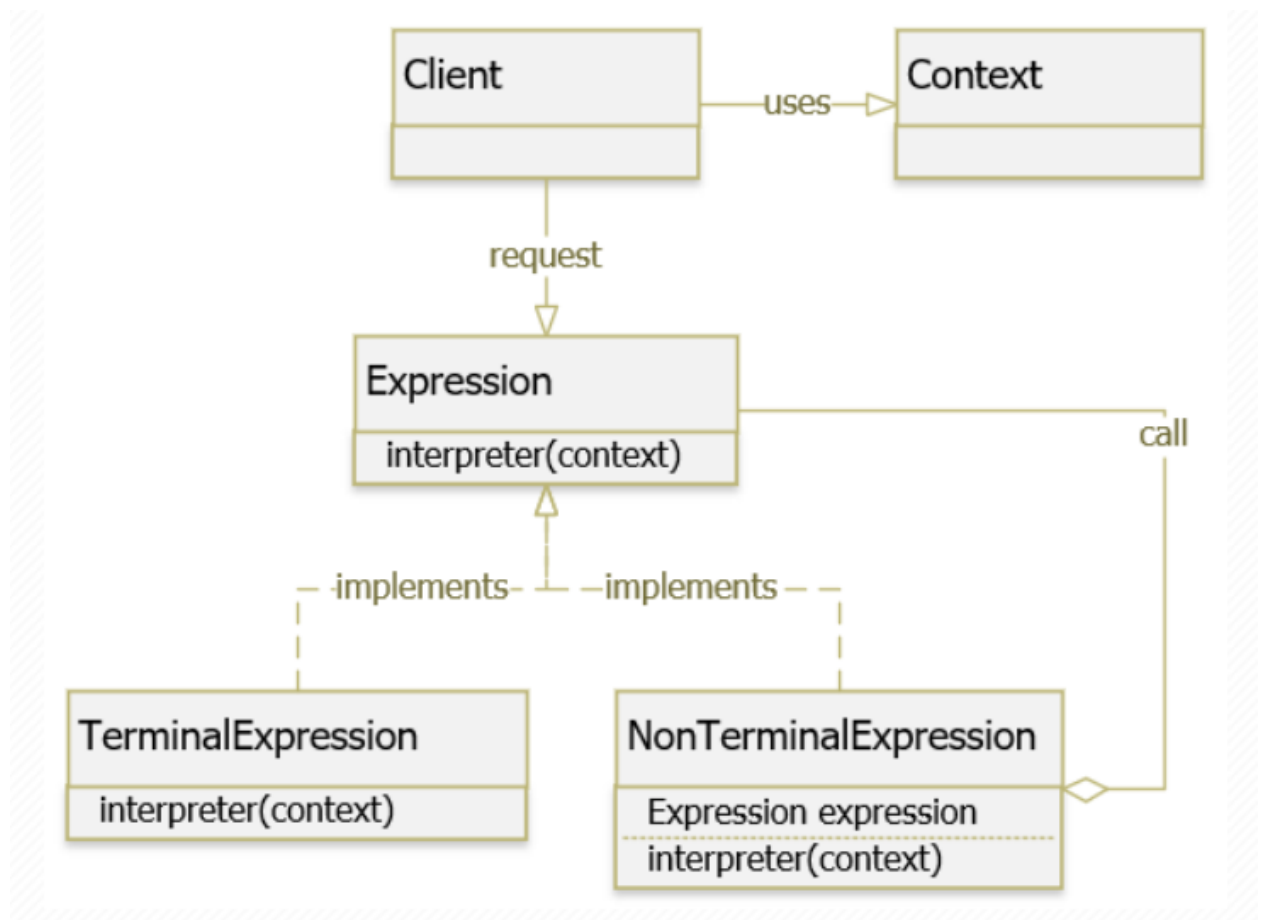
Interpreter nghĩa là thông dịch, mẫu này nói rằng “để xác định một biểu diễn ngữ pháp của một ngôn ngữ cụ thể, cùng với một thông dịch viên sử dụng biểu diễn này để diễn dịch các câu trong ngôn ngữ”. Nói cho dễ hiểu, Interpreter Pattern giúp người lập trình có thể “xây dựng” những đối tượng “động” bằng cách đọc mô tả về đối tượng rồi sau đó “xây dựng” đối tượng đúng theo mô tả đó.

Metadata (mô tả) -> [Interpreter Pattern] -> Đối tượng tương ứng.

Interpreter Pattern có hạn chế về phạm vi áp dụng. Mẫu này thường được sử dụng để định nghĩa bộ ngữ pháp đơn giản (grammar), trong các công cụ quy tắc đơn giản (rule),... Hoặc trường hợp cụ thể chúng ta hay gặp nhất là chuyển đổi ngôn ngữ trong menu của một ứng dụng : Open – mở, Exit – thoát, Copy – sao chép, Paste – Dán,...

Một ví dụ khác là khi chúng ta có một ngôn ngữ code phức tạp như java. Để in ra màn hình chúng ta cần gõ một câu lệnh rất dài : **System.out.println(“In ra màn hình”)**. Thay vào đó sử dụng Interpreter Pattern chúng ta chỉ cần ghi : **print(“In ra màn hình”)** và chúng ta vẫn có thể có được kết quả tương tự.

2.2.2 Cài đặt Interpreter Pattern



Các thành phần tham gia mẫu Interpreter:

- **Context** : là phần chứa thông tin biểu diễn mẫu chúng ta cần xây dựng.
- **Expression** : là một interface hoặc abstract class, định nghĩa phương thức `interpreter` chung cho tất cả các node trong cấu trúc cây phân tích ngữ pháp. **Expression** được biểu diễn như một cấu trúc cây phân cấp, mỗi implement của **Expression** có thể gọi một node.
- **TerminalExpression** (biểu thức đầu cuối): cài đặt các phương thức của **Expression**, là những biểu thức có thể được diễn giải trong một đối tượng duy nhất, chứa các xử lý logic để đưa thông tin của context thành đối tượng cụ thể.
- **NonTerminalExpression** (biểu thức không đầu cuối): cài đặt các phương thức của **Expression**, biểu thức này chứa một hoặc nhiều biểu thức khác nhau, mỗi biểu thức

có thể là biểu thức đầu cuối hoặc không phải là biểu thức đầu cuối. Khi một phương thức `interpret()` của lớp biểu thức không phải là đầu cuối được gọi, nó sẽ gọi đệ quy đến tất cả các biểu thức khác mà nó đang giữ.

- **Client** : đại diện cho người dùng sử dụng lớp Interpreter Pattern. Client sẽ xây dựng cây biểu thức đại diện cho các lệnh được thực thi, gọi phương thức `interpreter()` của node trên cùng trong cây, có thể truyền context để thực thi tất cả các lệnh trong cây.

2.2.3 Lợi ích của Interpreter Pattern

Dễ dàng thay đổi và mở rộng ngữ pháp. Vì mẫu này sử dụng các lớp để biểu diễn các quy tắc ngữ pháp, chúng ta có thể sử dụng thừa kế để thay đổi hoặc mở rộng ngữ pháp. Các biểu thức hiện tại có thể được sửa đổi theo từng bước và các biểu thức mới có thể được định nghĩa lại các thay đổi trên các biểu thức cũ.

Cài đặt và sử dụng ngữ pháp rất đơn giản. Các lớp xác định các nút trong cây cú pháp có các implement tương tự. Các lớp này dễ viết và các phân cấp con của chúng có thể được tự động hóa bằng trình biên dịch hoặc trình tạo trình phân tích cú pháp.

2.2.4 Sử dụng Interpreter Pattern khi nào

Interpreter Pattern được sử dụng hiệu quả khi:

- Bộ ngữ pháp đơn giản. Pattern này cần xác định ít nhất một lớp cho mỗi quy tắc trong ngữ pháp. Do đó ngữ pháp có chứa nhiều quy tắc có thể khó quản lý và bảo trì.
- Không quan tâm nhiều về hiệu suất. Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.

Interpreter Pattern thường được sử dụng trong trình biên dịch (compiler), định nghĩa các bộ ngữ pháp, rule, trình phân tích SQL, XML, ...

2.3 Ví dụ

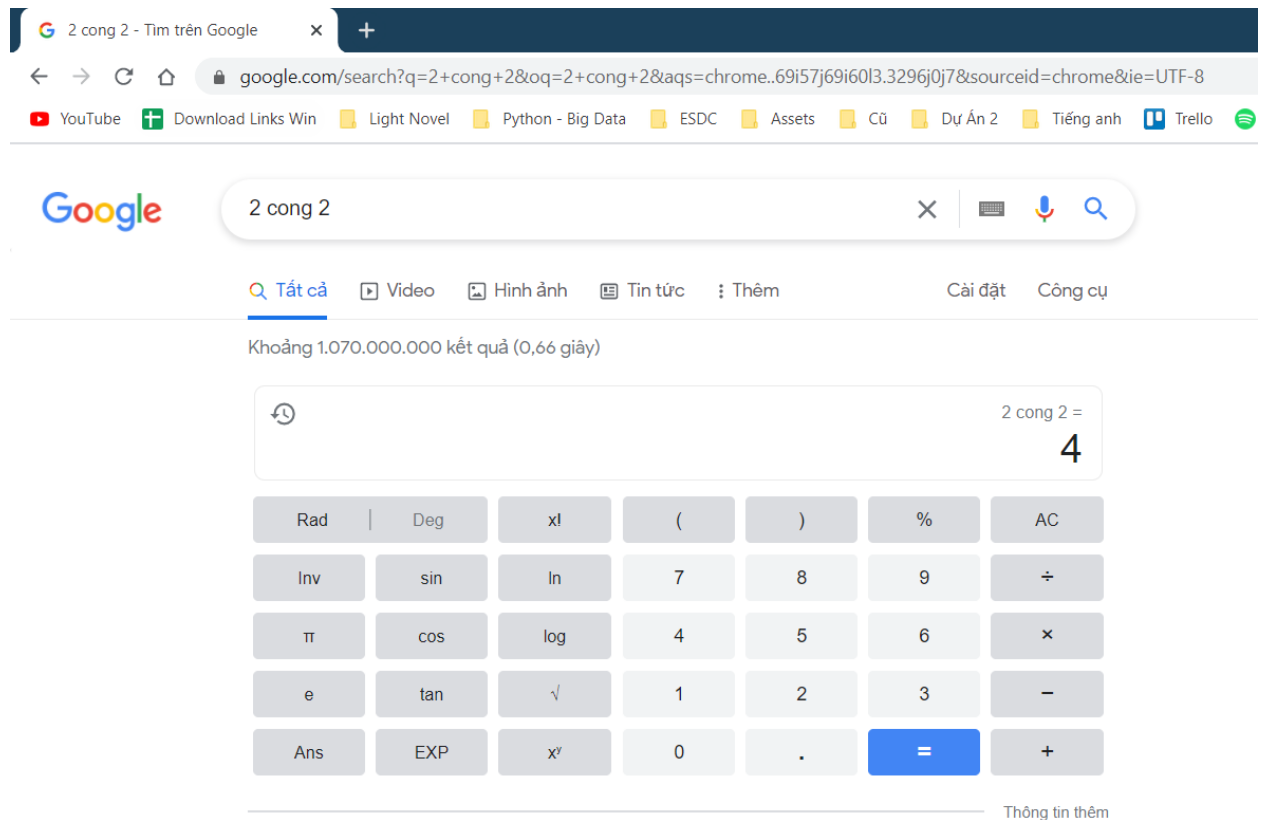
Xây dựng một ứng dụng tính toán cộng, trừ, căn bậc 2 và mũ 2 của một số nguyên.
Kết quả ra là một số nguyên.

Ví dụ :

- input : 2 cộng 2
- output : 4

- input : 2 mũ 3
- output : 8

Một ví dụ khá hay mọi người có thể thử là lên trình duyệt Chrome gõ : 2 cộng 2 và chúng ta sẽ có kết quả là 4



Tương tự như vậy với số mũ 2 :

2 mu 3 - Tìm trên Google

+

← → ↻ 🏠

google.com/search?q=2+mu+3&oq=2+mu+3&aqs=chrome.0.69i59j0i22i30l4j69i60l3.1886j0j7&sourceid=chrome&ie

📺 YouTube 📄 Download Links Win 📖 Light Novel 📄 Python - Big Data 📄 ESDC 📄 Assets 📄 Cũ 📄 Dự Án 2 📄 Tiếng anh 📄 Tre

Google

2 mu 3

✕ 🗑️ 🔊 🔍

🔍 Tất cả 📺 Video 📰 Tin tức 🖼️ Hình ảnh 🛒 Mua sắm ⋮ Thêm Cài đặt Công cụ

Khoảng 1.980.000.000 kết quả (0,47 giây)

🔄

2 mu 3 =

8

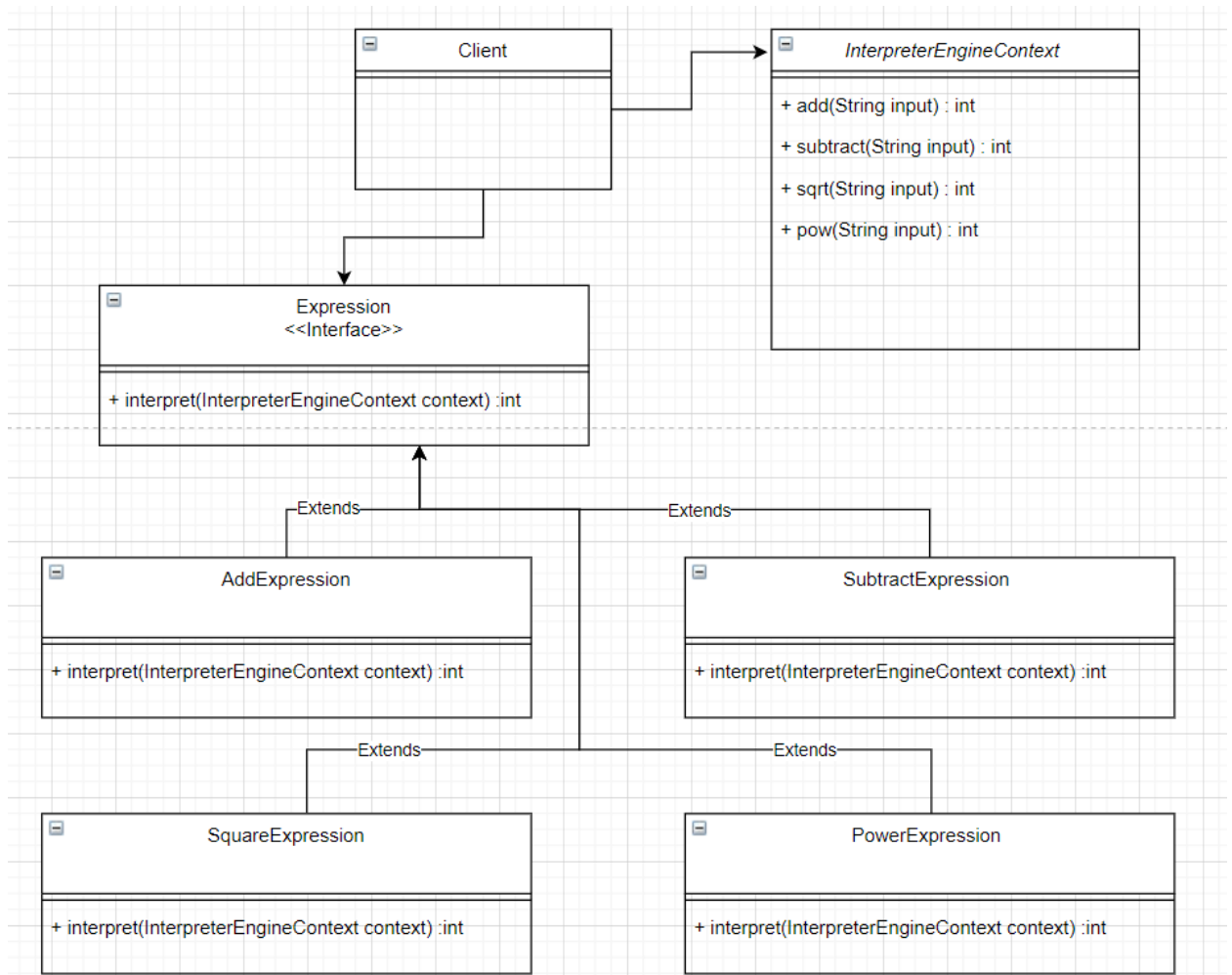
| | | | | | | |
|-----|-----|----------------|---|---|---|----|
| Rad | Deg | x! | (|) | % | AC |
| Inv | sin | ln | 7 | 8 | 9 | ÷ |
| π | cos | log | 4 | 5 | 6 | × |
| e | tan | √ | 1 | 2 | 3 | − |
| Ans | EXP | x ^y | 0 | . | = | + |

Thông tin thêm

2.4 Áp dụng

Sơ đồ class được vẽ trên <https://app.diagrams.net/> lưu tại file : documents/Bai tap lon - Mau thiet ke.drawio

Sơ đồ class :



Đầu tiên chúng ta cần xây dựng một context chứa thông tin và thực hiện.

```
3  public class InterpreterEngineContext {
4
5      public int add(String input) {
6          String[] tokens = interpret(input);
7          int num1 = Integer.parseInt(tokens[0]);
8          int num2 = Integer.parseInt(tokens[1]);
9          return (num1 + num2);
10     }
11
12     public int subtract(String input) {
13         String[] tokens = interpret(input);
14         int num1 = Integer.parseInt(tokens[0]);
15         int num2 = Integer.parseInt(tokens[1]);
16         return (num1 - num2);
17     }
18
19     public int sqrt(String input) {
20         String[] tokens = interpret(input);
21         int num = Integer.parseInt(tokens[0]);
22         return (int)Math.sqrt(num);
23     }
24
25     public int pow(String input) {
26         String[] tokens = interpret(input);
27         int num1 = Integer.parseInt(tokens[0]);
28         int num2 = Integer.parseInt(tokens[1]);
29
30         return (int)Math.pow(num1,num2);
31     }
32
33     private String[] interpret(String input) {
34         String str = input.replaceAll("[^0-9]", " ");
35         str = str.replaceAll("( )+", " ").trim();
36         return str.split(" ");
37     }
38 }
```


Tiếp đến là interface định nghĩa interpreter chung cho tất cả các node trong cấu trúc phân tích ngữ pháp. Và node ở đây là các class đã implement interface này :

```
1 public interface Expression {  
2     int interpret(InterpreterEngineContext context);  
3 }
```

Bốn class cộng, trừ, mũ, căn bậc 2 implement interface bên trên.

```
1 public class AddExpression implements Expression {  
2  
3     private String expression;  
4  
5     public AddExpression(String expression) {  
6         this.expression = expression;  
7     }  
8  
9     @Override  
10    public int interpret(InterpreterEngineContext context) {  
11        return context.add(expression);  
12    }  
13 }
```

```
1 public class SubtractExpression implements Expression {  
2  
3     private String expression;  
4  
5     public SubtractExpression(String expression) {  
6         this.expression = expression;  
7     }  
8  
9     @Override  
10    public int interpret(InterpreterEngineContext context) {  
11        return context.subtract(expression);  
12    }  
13 }
```

```
1 public class PowerExpression implements Expression {
2
3     private String expression;
4
5     public PowerExpression(String expression) {
6         this.expression = expression;
7     }
8
9     @Override
10    public int interpret(InterpreterEngineContext context) {
11        return context.pow(expression);
12    }
13 }
```

```
1 public class SquareExpression implements Expression {
2
3     private String expression;
4
5     public SquareExpression(String expression) {
6         this.expression = expression;
7     }
8
9     @Override
10    public int interpret(InterpreterEngineContext context) {
11        return context.sqrt(expression);
12    }
13 }
```

Cuối cùng là Client sẽ gọi đến và sử dụng:

```
1 public class Client {
2
3     Run | Debug
4     public static void main(String args[]) {
5         System.out.println("2 cong 2 = " + interpret("2 cong 2"));
6         // System.out.println("10 tru 4 = " + interpret("10 tru 4"));
7         System.out.println("2 mu 3 = " + interpret("2 mu 3"));
8         System.out.println("can 4 = " + interpret("can 4"));
9     }
10
11     private static int interpret(String input) {
12         Expression exp = null;
13         if (input.contains("cong")) {
14             exp = new AddExpression(input);
15         } else if (input.contains("tru")) {
16             exp = new SubtractExpression(input);
17         } else if (input.contains("mu")) {
18             exp = new PowerExpression(input);
19         } else if (input.contains("can")) {
20             exp = new SquareExpression(input);
21         } else {
22             throw new UnsupportedOperationException();
23         }
24         return exp.interpret(new InterpreterEngineContext());
25     }
26 }
```

Code được đặt tại : .\source\Interpreter Pattern

Các bước chạy :

- javac *.java
- java Client

Kết quả :

```
C:\Hoctap\MauThietKe\doan\Mau-thiet-ke-doan\Baitaplon\source\Interpreter Pattern>javac *.java

C:\Hoctap\MauThietKe\doan\Mau-thiet-ke-doan\Baitaplon\source\Interpreter Pattern>java Client
2 cong 2 = 4
2 mu 3 = 8
can 4 = 2
```

2.5 Kết luận

Interpreter Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Interpreter nghĩa là thông dịch, mẫu này nói rằng “để xác định một biểu diễn ngữ pháp của một ngôn ngữ cụ thể, cùng với một thông dịch viên sử dụng biểu diễn này để diễn dịch các câu trong ngôn ngữ”.

Interpreter Pattern được sử dụng hiệu quả khi:

- Bộ ngữ pháp đơn giản.
- Không quan tâm nhiều về hiệu suất.

Interpreter Pattern thường được sử dụng trong trình biên dịch (compiler), định nghĩa các bộ ngữ pháp, rule, trình phân tích SQL, XML, ...

TÀI LIỆU THAM KHẢO

1. <https://viblo.asia/p/tong-hop-cac-bai-huong-dan-ve-design-pattern-23-mau-co-ban-cua-gof-3P0lPQPG5ox>
2. <https://tuhocict.com/chain-of-responsibility-chuoi-xu-ly-yeu-cau/>
3. <https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/>
4. <https://www.programiz.com/java-programming/enum-constructor>
5. <https://viblo.asia/p/design-patterns-chain-of-responsibility-pattern-bJzKm1qwK9N>
6. <https://v1study.com/design-patterns-mau-chain-of-responsibility.html>