TRƯỜNG ĐẠI HỌC QUY NHƠN

KHOA CÔNG NGHỆ THÔNG TIN

-----□□□□□-----

BÁO CÁO HỌC PHẦN: THIẾT KẾ PHẦN MỀM

TOPIC : INTERPRETER PATTERN

Nhóm 11: Lê Văn Đại

Trần Tuệ Khánh

Bùi Thị Diệu

Giảng viên: Phạm Văn Việt

Mục lục

1. Tổng quan

a) Interpreter Pattern

- Interpreter Pattern là một mẫu thiết kế thuộc nhóm hành vi và được sử dụng để xác định biểu diễn ngữ pháp cho một ngôn ngữ và cung cấp trình thông dịch để xử lý ngữ pháp này.
- Mẫu này liên quan đến việc triển khai một giao diện biểu thức cho phép diễn giải một ngữ cảnh cụ thể. Mẫu này được sử dụng trong phân tích cú pháp SQL, công cụ xử lý ký hiệu,...
- Mẫu này tổ chức các biểu thức theo hệ thống cấp bậc: mỗi biểu thức là biểu thức đầu cuối (terminal) hoặc biểu thức không đầu cuối (non-terminal)
- Cấu trúc cây của *Interpreter* tương tự *Composite*: biểu thức đầu cuối (terminal) là các nút lá, còn biểu thức không đầu cuối (non-terminal) là các nút hợp chất.
- Cây biểu thức cần đánh giá thường được tạo bởi trình phân tích cú pháp (parser); bản thân parser không thuộc phạm vi của Interpreter Pattern.

b) Khi nào nên sử dụng Interpreter Pattern

Sử dụng Interpreter Pattern khi chúng ta muốn:

- 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, …

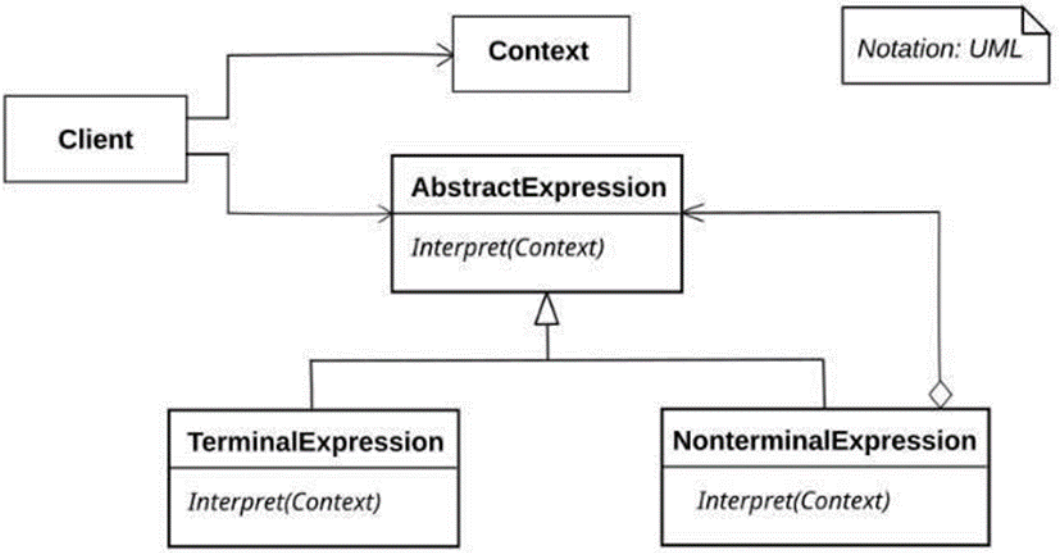
c) Khi nào không nên dùng

Bởi một số lý do sau:

- Khi có các phép tính hoặc thao tác đơn giản.
- Khi ngữ pháp phức tạp.

- Yêu cầu hiệu suất cao.

2. Kiến trúc



Các thành phần trong mô hình:

- **AbstractExpression**: Khai báo một giao diện cho việc thực hiện một thao tác.
- **TerminalExpression**: Cài đặt một thao tác thông dịch liên kết với những ký pháp đầu cuối, đóng vai trò một thể nghiệm được yêu cầu cho mọi ký pháp đầu cuối trong câu.
- **NonterminalExpression**: Có thể chứa TerminalExpression bên trong và cũng có thể chứa một NonterminalExpression khác. Nó đóng vai trò như là "ngữ pháp" của ngôn ngữ đặc tả.
- **Context**: Là đối tượng thông tin để thực hiện thông dịch. Đối tượng này là toàn cục đối với quá trình thông dịch (dùng chung giữa các node).
- **Client** (ExpressionParser): là thành phần đại diện cho người dùng của *Interpreter Pattern*. Client xây dựng cây biểu thức (AST) đại diện cho các lệnh cần thực thi, sau đó gọi phương thức Interpret() trên nút gốc của cây, truyền Context nếu cần để thực thi toàn bộ lệnh trong cây.

3. Bài toán mà Interpreter Pattern giải quyết

- Vấn đề đưa ra đó là: Trong nhiều hệ thống phần mềm, chúng ta cần xử lý, phân tích và thực thi các câu lệnh hoặc biểu thức được mô tả bằng một ngôn ngữ riêng(domain-specific language - DSL).

Các biểu thức này có thể là:

- Các lệnh điều kiện (ví dụ: department == HR OR level > 5)
- Biểu thức toán học (2+ 3 +5)
- Câu truy vấn đơn giản (SELECT NAME FROM EMPLOYEES WHERE SALARY > 5000)
- Quy tắc nghiệp vụ có thể thay đổi thường xuyên

Ngoài những cái nêu trên còn nhiều các biểu thức khác, chủ yếu là chúng không quá phức tạp cầu kỳ, nhưng lại dễ thay đổi.

Vậy vấn đề đưa ra là: Làm thế nào để hệ thống có thể hiểu, diễn giải và thực thi những biểu thức đó một cách linh hoạt, có thể mở rộng, mà không cần viết lại toàn bộ mã lệnh mỗi khi quy tắc thay đổi?

Để giải quyết vấn đề trên, Interpreter Pattern cung cấp một cấu trúc hướng đối tượng để biểu diễn và thực thi các câu lệnh hoặc biểu thức. Bằng cách biểu diễn quy tắc của ngữ pháp thành một lớp đối tượng riêng biệt và xây dựng cây cú pháp trừu tượng, hệ thống có thể đọc hiểu, mở rộng, và thực thi các biểu thức một cách linh hoạt, có cấu trúc và dễ bảo trì.

Ý tưởng chính là:

1. Xác định một ngữ pháp (grammar) cho ngôn ngữ cần diễn giải.

2. Xây dựng các lớp biểu diễn từng thành phần của ngữ pháp (số, phép toán, toán tử logic, biến...).

3. Tạo cây cú pháp trừu tượng (Abstract Syntax Tree – AST) từ biểu thức đầu vào.

4. Gọi phương thức interpret() trên nút gốc của cây để diễn giải toàn bộ biểu thức.

Khi đó:

- Mỗi lớp trong cây chịu trách nhiệm tự hiểu và thực thi phần ngữ pháp của mình.
- Ta có thể kết hợp, mở rộng hoặc thay thế các biểu thức mà không cần sửa logic chung.

4. Ví dụ bài toán cụ thể: Xây Dựng Máy Tính Biểu Thức Đơn Giản

Để minh họa, chúng ta sẽ xây dựng một ứng dụng có khả năng tính toán giá trị của các biểu thức số học chỉ bao gồm các con số nguyên cùng với hai phép toán là **cộng (+)** và **trừ (-)**.

- **Ngôn ngữ**: Biểu thức số học (ví dụ: "100 - 20 + 5").
- **Ngữ pháp**:
    - **Biểu thức cuối (Terminal Expression)**: Là các số nguyên (ví dụ: 100, 20, 5).
    - **Biểu thức không cuối (Non-terminal Expression)**: Là các phép toán kết hợp hai biểu thức khác (ví dụ: biểu_thức_1 + biểu_thức_2).
- **Mục tiêu**: Chương trình nhận một chuỗi biểu thức và trả về kết quả số nguyên cuối cùng.

Dưới đây là mã nguồn minh họa bằng java:

```
// File: Expression.java

/**

 * 1. AbstractExpression: Định nghĩa giao diện chung cho tất cả các nút trong cây cú pháp.

 */

interface Expression {

    int interpret();

}


// File: NumberExpression.java

/**

 * 2. TerminalExpression: Đại diện cho một con số (một "lá" trong cây).

 */

class NumberExpression implements Expression {

    private int number;

    public NumberExpression(String number) {

        this.number = Integer.parseInt(number);

            }

    @Override

    public int interpret() {

        return this.number;

            }

}
```

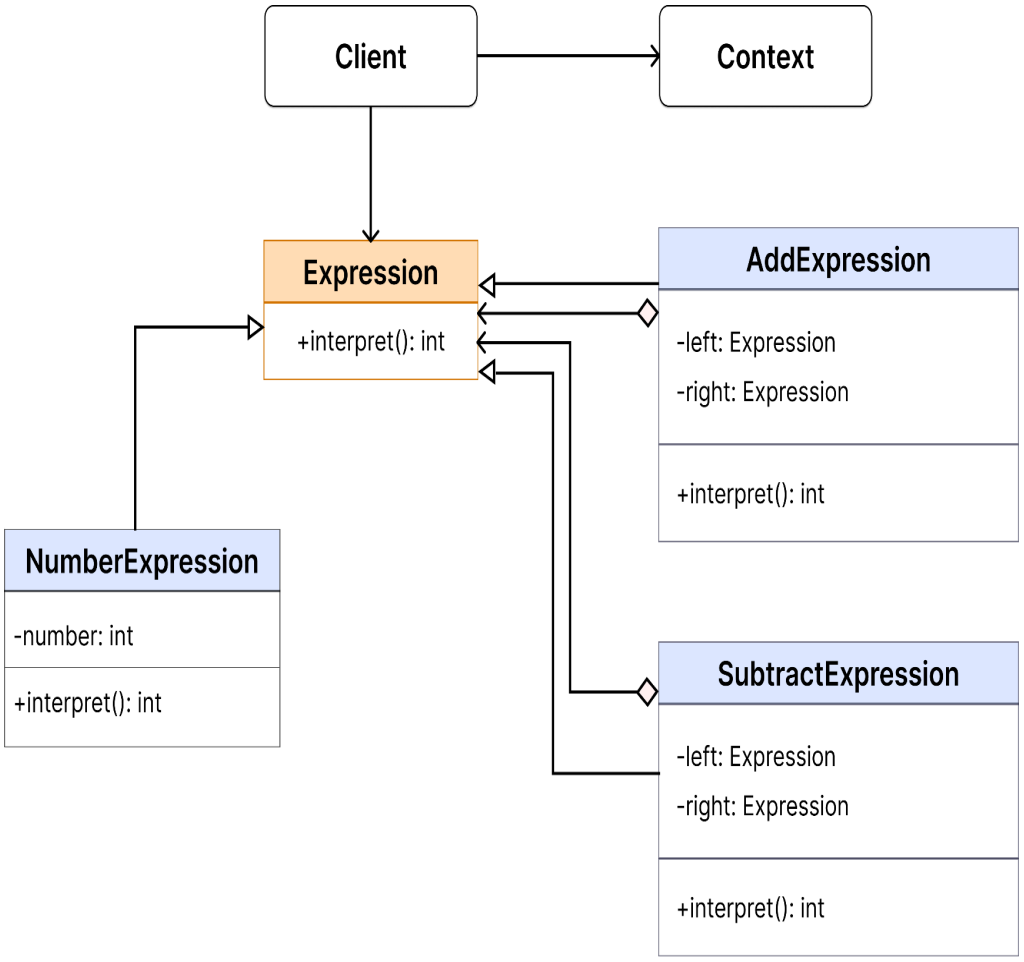
```
// File: AddExpression.java
/**
 * 3. NonTerminalExpression: Đại diện cho phép toán cộng.
 */
class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
            }
    @Override
    public int interpret() {
        // Gọi đệ quy interpret() trên các biểu thức con và thực hiện phép cộng
        return left.interpret() + right.interpret();
            }
}
```

```
// File: SubtractExpression.java
/**
 * 4. NonTerminalExpression: Đại diện cho phép toán trừ.
 */
class SubtractExpression implements Expression {
    private Expression left;
    private Expression right;

    public SubtractExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
           }

    @Override
           public int interpret() {
        // Gọi đệ quy interpret() trên các biểu thức con và thực hiện phép trừ
        return left.interpret() - right.interpret();
           }
}
```

```
// File: CalculatorClient.java

/**

 * 5. Client: Xây dựng cây cú pháp và thực thi.

 * Trong thực tế, việc phân tích chuỗi và xây dựng cây sẽ được thực hiện bởi một lớp Parser.

 * Ở đây, chúng ta xây dựng cây thủ công để minh họa.

 */

public class CalculatorClient {


    public static void main(String[] args) {

        // Bài toán: Tính giá trị của biểu thức "10 + 5 - 3"

        // 1. Xây dựng cây cú pháp trừu tượng (AST)

        // Cây được xây dựng tương ứng với thứ tự ưu tiên: (10 + 5) - 3

        Expression expressionTree = new SubtractExpression(

            new AddExpression(

                    new NumberExpression("10"),

                    new NumberExpression("5")

            ),

            new NumberExpression("3")

        );


        // 2. Bắt đầu quá trình thông dịch từ nút gốc của cây

        int result = expressionTree.interpret();
```

```
        System.out.println("Giá trị của biểu thức '10 + 5 - 3' là: " + result);
        // Kết quả mong muốn: 12
    }
}
```

5. Sơ đồ UML mô tả thiết kế



Các thành phần trong mô hình:

- **AbstractExpression** (Giao diện biểu thức trừu tượng):
    + Tên thành phần: Expression.
    + Mô tả: Đây là một giao diện xác định một phương thức interpret() chung mà tất cả các biểu thức cụ thể phải triển khai. Phương thức này chịu trách nhiệm thông dịch và trả về kết quả.
- **TerminalExpression** (Biểu thức cuối)
    + Tên thành phần: NumberExpression.
    + Mô tả: Đây là các lớp cụ thể đại diện cho các phần tử cơ bản (hoặc "lá") trong ngữ pháp của ngôn ngữ. Trong mã, NumberExpression đại diện cho một con số, là một phần tử cuối cùng không thể phân tách thêm trong cây cú pháp. Phương thức interpret() của nó chỉ đơn giản là trả về giá trị số của nó.
- **NonTerminalExpression** (Biểu thức không cuối)
    + Tên thành phần: AddExpression và SubtractExpression.
    + Mô tả: Đây là các lớp cụ thể đại diện cho các quy tắc phức tạp hơn trong ngữ pháp. Chúng được xây dựng từ các Expression khác (cả TerminalExpression và các NonTerminalExpression khác), tạo thành một cấu trúc cây.
- AddExpression thực hiện phép cộng trên các biểu thức con của nó.
- SubtractExpression thực hiện phép trừ trên các biểu thức con của nó.
- Phương thức interpret() của chúng gọi đệ quy phương thức interpret() trên các biểu thức con để tính toán kết quả cuối cùng.
- **Client** (Khách hàng)
    + Tên thành phần: CalculatorClient.

    + Mô tả: Đây là lớp tạo ra cây cú pháp trừu tượng (Abstract Syntax Tree - AST) bằng cách sử dụng các đối tượng NonTerminalExpression và TerminalExpression. Sau đó, nó gọi phương thức interpret() tại nút gốc của cây để bắt đầu quá trình thông dịch và lấy kết quả. Trong ví dụ này, CalculatorClient xây dựng cây cho biểu thức "10 + 5 - 3" và bắt đầu thông dịch.

6. Lợi ích khi dùng Interpreter Pattern

- **Mở rộng dễ dàng (OCP)**: thêm quy tắc/toán tử mới chỉ bằng cách tạo lớp Expression mới, không chạm nhiều vào mã cũ.
- **Cấu trúc rõ ràng (AST)**: logic được biểu diễn dạng cây biểu thức → dễ đọc, dễ hình dung luồng đánh giá.
- **Tách biệt mối quan tâm**: ngữ pháp/biểu thức tách khỏi mã nghiệp vụ, Context giữ dữ liệu, Expression lo diễn giải.
- **Tái sử dụng & phối hợp**: các biểu thức nhỏ (terminal) có thể kết hợp (AND/OR/NOT, ADD/MUL…) để tạo quy tắc phức tạp.
- **Dễ kiểm thử**: test từng nút biểu thức độc lập (unit test) rồi test tổ hợp (integration) trên cây lớn.
- **Cấu hình hóa quy tắc**: có thể cho phép DSL đơn giản để người dùng/BA chỉnh quy tắc mà không sửa code.
- **Nhất quán hành vi**: cùng một cây biểu thức có thể áp dụng cho nhiều bản ghi/ngữ cảnh khác nhau, đảm bảo kết quả nhất quán.
- **Tương thích mẫu khác**: hình thái cây gần Composite, có thể kết hợp **Builder/Factory** để dựng AST gọn hơn; dễ gắn với Specification cho filter.
- **Khả chuyển ngữ cảnh**: cùng ngữ pháp có thể dùng ở nhiều nơi (validate form, lọc dữ liệu, rule giá…), chỉ thay Context.