TÌM HIỂU VỀ DESIGN PATTERN

FLYWEIGHT VÀ INTERPRETER PATTERN

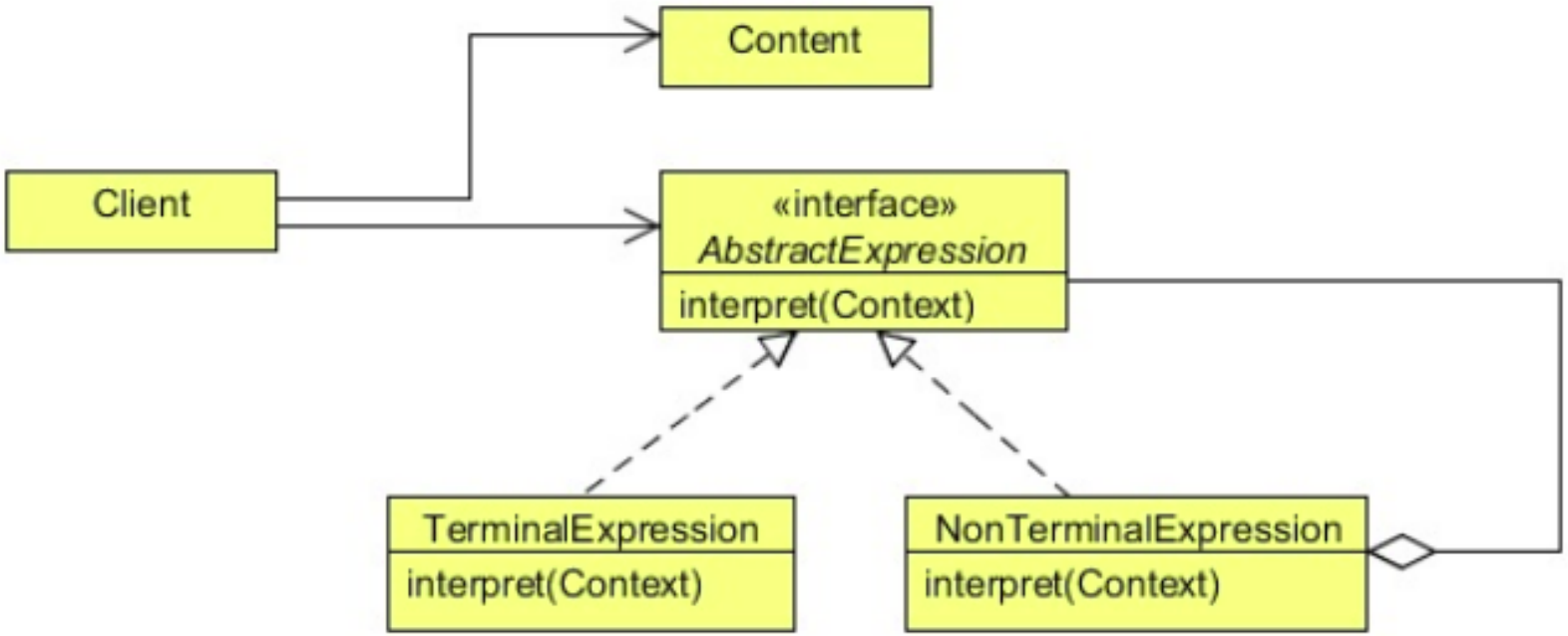
Sinh viên: Đặng Lâm San - 20170111

Ngày 7 tháng 12 năm 2020

Interpreter - Introduction

- Interpreter pattern is a behavioral pattern.
- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- In general, languages are made up of a set of grammar rules. Different sentences can be constructed by following these grammar rules.
- A simple example of this would be the set of different arithmetic expressions submitted to a calculator program.

Interpreter - Class diagram



Interpreter - Class diagram

- **AbstractExpression:**
  - Declares an abstract `interpret` operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression:**
  - Implements an `interpret` operation associated with terminal symbols in the grammar.
- **NonterminalExpression:**
  - Implements an `interpret` operation for non terminal symbols in the grammar. `interpret` typically calls itself recursively.
- **Context:**
  - Contains information that's global to the interpreter.
- **Client**
  - Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
  - Invokes the `interpret` operation.

Implementation of Interpreter pattern - Postfix expression

- **AbstractExpression**

```
package com.javacodegeeks.patterns.interpreterpattern;

public interface Expression {
        public int interpret();
}
```

Implementation of Interpreter pattern - Postfix expression

- **AddExpression**

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Add implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Add(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() + rightExpression.interpret();
        }

}
```

Implementation of Interpreter pattern - Postfix expression

- **ProductExpression**

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Product implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Product(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
                return leftExpression.interpret() * rightExpression.interpret();
        }
}
```

Implementation of Interpreter pattern - Postfix expression

- **SubtractExpression**

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Substract implements Expression{

        private final Expression leftExpression;
        private final Expression rightExpression;

        public Substract(Expression leftExpression,Expression rightExpression ){
                this.leftExpression = leftExpression;
                this.rightExpression = rightExpression;
        }
        @Override
        public int interpret() {
```

Java Design Patterns

```
                return leftExpression.interpret() - rightExpression.interpret();
        }

}
```

Implementation of Interpreter pattern - Postfix expression

- **NumberExpression**

```
package com.javacodegeeks.patterns.interpreterpattern;

public class Number implements Expression{

        private final int n;

        public Number(int n){
                this.n = n;
        }
        @Override
        public int interpret() {
                return n;
        }

}
```

Implementation of Interpreter pattern - Postfix expression

- **ExpressionUtils**

```
package com.javacodegeeks.patterns.interpreterpattern;

public class ExpressionUtils {

        public static boolean isOperator(String s) {
                if (s.equals("+") || s.equals("-") || s.equals("*"))
                        return true;
                else
                        return false;
        }

        public static Expression getOperator(String s, Expression left, Expression right) {
                switch (s) {
                case "+":
                        return new Add(left, right);
                case "-":
                        return new Substract(left, right);
                case "*":
                        return new Product(left, right);
                }
                return null;
        }

}
```

Implementation of Interpreter pattern - Postfix expression

- **Client**

```
package com.javacodegeeks.patterns.interpreterpattern;

import java.util.Stack;

public class TestInterpreterPattern {

        public static void main(String[] args) {

                String tokenString = "7 3 - 2 1 + *";
                Stack<Expression> stack = new Stack<>();
                String[] tokenArray = tokenString.split(" ");
                for (String s : tokenArray) {
                        if (ExpressionUtils.isOperator(s)) {
                                Expression rightExpression = stack.pop();
                                Expression leftExpression = stack.pop();
                                Expression operator = ExpressionUtils.getOperator(s,
                                    leftExpression,rightExpression);
                                int result = operator.interpret();
                                stack.push(new Number(result));
                        } else {
                                Expression i = new Number(Integer.parseInt(s));
                                stack.push(i);
                        }
                }
                System.out.println("( "+tokenString+" ): "+stack.pop().interpret());

        }
```

When to use Interpreter?

- Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when:
  - The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable.
  - Efficiency is not a critical concern.