Object-oriented Realization of Mathematical Expression Based on Interpreter

Jing Shi [1,a,*], Bing Ren [2,b]

[1]Xianyang vocational and technical college, Xianyang, Shaanxi, China

[2]China Construction Bank Shaanxi Branch, Xi'an, Shaanxi, China

[a] 258587000@qq.com, [b] 54044805@qq.com

[*]corresponding author

Keyword: Object-oriented**;** Interpreter model; Mathematical Expression

Abstract: In this paper, the basic concept of design pattern is introduced at first. Through the detailed introduction of the behavior pattern of Interpreter pattern, an object-oriented realization of mathematical expression is designed by using it. At last, the advantages and disadvantages of the Interpreter pattern are summarized.

1. Design Pattern Overview

Design pattern is a combination of methodology and a group of abstract patterns used to design reusable object-oriented software in the field of object-oriented software design in recent years. Design pattern is a very powerful tool, which uses the accumulated knowledge and experience of programmers to solve the programming problems that people often encounter, and provides a high-level scheme that is easy to reuse and maintainable. If we use design pattern flexibly, it can save us a lot of time. A design pattern provides a solution to a specific problem. In object-oriented programming, a design pattern has a specific structure. With this structure, the program architecture is more flexible, the code can be reused and maintained more easily, and the program has better adaptability.

Design patterns make it more convenient for designers to improve or reuse some past mature designs and architectures. It will be proved by practice that it will make it easier for new system developers to understand their design ideas. As long as designers understand the design pattern, they can absorb the valuable experience in the pattern to a large extent, and have a better understanding of the object-oriented system. At the same time, these patterns can be directly used to guide the critical modeling problems in object-oriented systems. If you have the same problem background, you can apply it directly. More specifically, through the use of design patterns, it can enhance the reuse function of the packaged classes, and effectively handle the change of requirements. Some patterns can reduce the coupling and dependence between classes. More importantly, designers familiar with design patterns can give them new design ideas.

2. Interpreter --- Behavior Pattern of Class

The interpreter is the behavior pattern of a class. Given any language, the explain mode can define a indication of its grammar and provide an interpreter, which uses the indication to interpret the sentences in the language. The client can use this interpreter to interpret sentences in this language. In the event of a specially appointed mold of problem happens frequently, it may be deserve expressing each instance of the problem as a sentence in a simple language. This allows you to build an interpreter, this interpreter is used to interpret these sentences, so as to solve such problems. How to define a grammar for a simple language can be described by an interpreter pattern,The Interpreter pattern describes how to define a grammar for a simple language, and how to express a sentence in the language, and how to interpret these sentences.

Composition of Interpreter pattern:

1) Abstract expression role: declare an abstract interpretation operation, which is implemented by

all concrete expression roles (nodes in the abstract syntax tree).

Each node of the abstract syntax tree represents a statement, and interpretation methods can be executed on each node. The execution of this interpretation method represents that the statement is interpreted. Because each statement is interpreted on behalf of this statement. Since each statement represents an instance of a common problem, the explanation operation on each node represents an answer to a problem instance.

2) Terminator expression role: specific expression.

a) Implement the interpretation operation associated with the terminator in grammar

b) And each terminator in a sentence requires an instance of the class to correspond to it

3) Nonterminal expression role: concrete expression.

a) Every rule in grammar R::=R1R2…Rn requires a non terminator table band role

b) Maintain an instance variable for the abstract expression role for each symbol from R1 to RN

c)To implement the interpretation operation, it is generally necessary to call the interpretation operation representing the objects from R1 to RN recursively

4) Context (environment) role: contains some global information outside the interpreter.

5) Customer roles:

a) Construct (or be given) an abstract syntax tree that represents a specific sentence in the grammar defined language

b) Call interpretation operation

Interpreter mode has its applicability: When you need to interpret and execute a language and represent sentences in that language in an abstract syntax tree, you can use the pattern of interpreter . When existing the following conditions this mode effects best :

• For complex grammar, the class level of grammar becomes huge and cannot be managed. At this point, a tool like parser generator is a better choice. In this way, the abstract syntax tree can interpret the corresponding expressions without being built, which can save a lot of time and space.

• The high efficiency interpreter for a key problem is general not implemented by directly explaining the parsing tree, but at first converting them into another kind of shape. For example, this form of state machine is transformed by regular expression. Even though in this case, the converter can still be realized in the mode of interpreter, which is still useful.

3. Realization of Mathematical Expression Through Interpreter Pattern

Each grammar rule of the Interpreter pattern is represented by a class. Regular expression is a standard language for describing string patterns. Instead of constructing a specific algorithm for each pattern, it is better to use a general search algorithm to interpret and execute a regular expression defines be matched. We also use regular expressions to express mathematical expressions. We define grammar and use Interpreter pattern to complete the object-oriented implementation of mathematical expressions.

When the mathematical expression is implemented through interpreter mode, the grammar is defined as follows:

expression ::= variableexp | literal | addexp | subexp | mulexp | divexp | extract rootexp | squexp | absolute valueexp `(`expression`)`

addexp ::= expression `add` expression

subexp ::= expression `sub` expression

mulexp ::= expression `mul` expression

divexp ::= expression `div` expression

extract rootexp ::= expression `extract root` expression

squexp ::= expression `squ` expression

literal ::= `1` | `2` | `3` |…

variableexp ::= `A` | `B` | … | `X` | `Y` | `Z`

In the grammar above, the symbol expression is the start symbol and the literal is the terminator that defines a simple word. In this language, the terminators are numbers, i.e. 1, 2, 3, etc. A

nonterminal represents a regular expression that contains the operators add, sub, Mul, div, extract root, squ, that is, add, subtract, multiply, divide, square and square. Some variables are also nonterminal.

Let's now define two operations on a regular expression. One operation is evaluate, which is to evaluate a mathematical expression in a context. Of course, the context must assign a specific number to every alternating quantity. Replace is the two operation, which replaces variable with an expression to produce new mathematical expression. The substitution operation shows that the Interpreter pattern can be used not only to find the value of an expression, but also for other purposes. Here, it is used to operate on the expression itself.

Here we have designed a total of nine classes, an abstract class expression and its seven subclasses addexp, subexp, mulex, divexp, extract rootexp, squaxp, variableexp.

The class diagram of the above-mentioned realization of mathematical expression through interpreter mode is shown as follows:

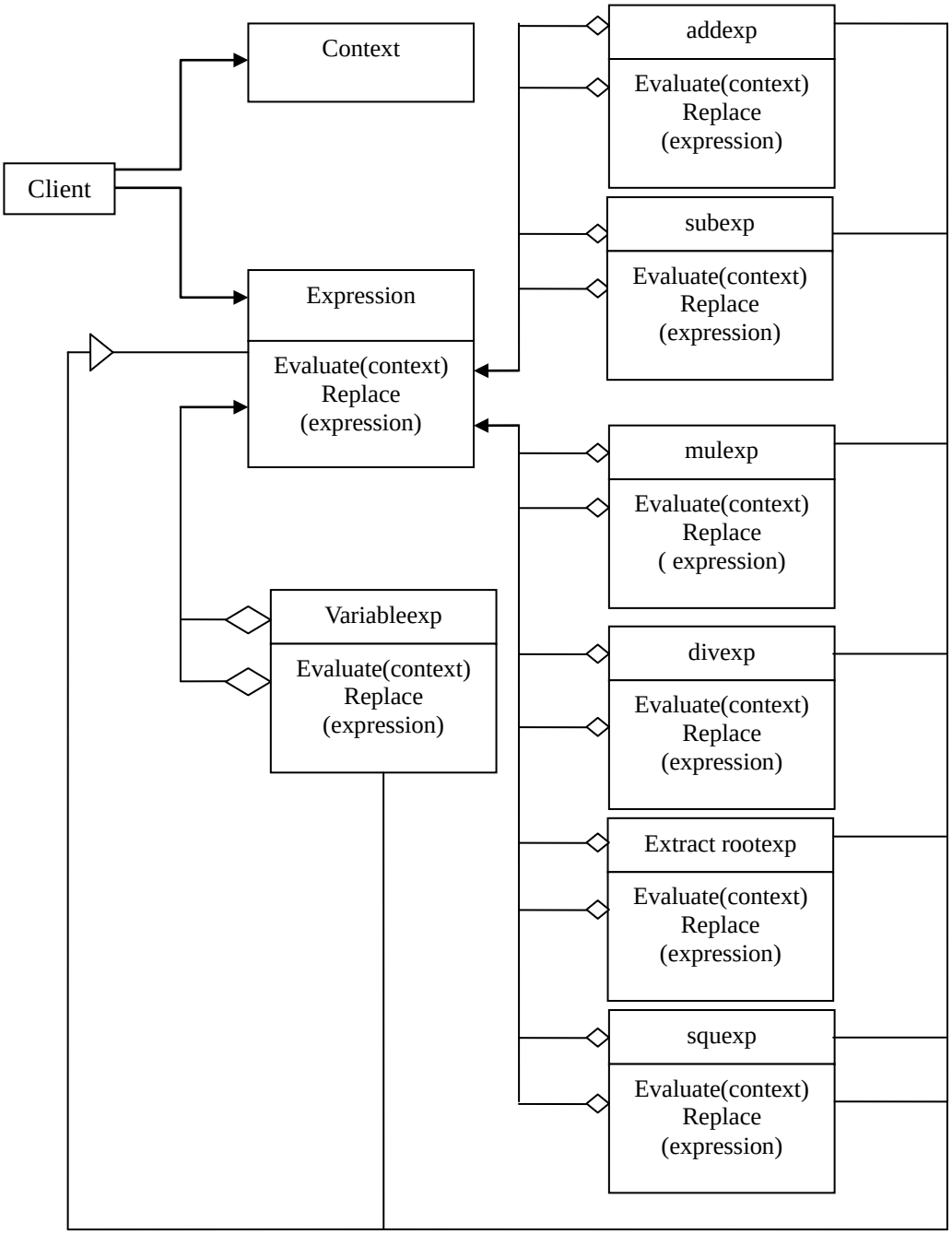

Figure 1 Mathematical expression through interpreter mode class diagram

Expression defines an interface for a mathematical expression:

```
class expression {
public:
expression();
        virtual ~expression();

        virtual exp evaluate(context) = 0;
        virtual expression* replace(literal char*, expression) = 0;
        virtual expression* copy() literal = 0;
};
```

Class context delimits a cast light upon from variables to results, which are represented by constants, i.e. some numbers. Context has interfaces:

```
class Context {
public:
        exp lookup(literal char*) literal;
        void assign(variableexp*, exp);
};
```

A variable means a nomenclature variable:

```
class varialbleexp : public expression {
public:
        variableexp(literal char*);
        virtual ~variableexp();

        virtual exp evaluate(context);
        virtual expression* replace(literal char*, expression);
        virtual expression* copy() literal;
private:
        char* _name;
};
```

The detailed definition of other concrete operation classes will not be explained here.

The detailed definition of other concrete operation classes will not be explained here. The above example shows an important feature of the interpreter mode: you can "interpret" a sentence with multiple operations. In the expression operation defined, evaluate can return a simple result and explain the corresponding program or expression. But, the replace operation can also be considered an interpreter. The name of the replacement variable and its expression can be used as the context of this interpreter, the new expression can be a result of it.

Through the interaction of these classes, we can use these classes to complete the operation of mathematical expression, and fully reflect the role of interpreter mode.

4. Interpreter Mode Summary

Interpreter mode has some advantages and disadvantages:

1) Grammar is easy to change and expand: Because grammar rules for interpreting patterns are represented by classes, class inheritance can be used to extend and change the grammar. It can be incremental changed by the engaged expression, while the new expression can be used of a variation of the old one.

2) It is also easy to implement Grammar: the implementation of the classes that define the nodes is similar. They can easy and directly to write these classes, and they can also use compiler or a parser generator to generate automatically.

3) Hard to maintain the Complex grammar: more classes are needed for grammar rules defined

by BNF. As a result, it is difficult to manage or maintain a grammar that contains many rules.To alleviate this kind of problem, some other design patterns can be adopted. If grammar is particularly complex, technical support such as parser or compiler generator is more appropriate..

4) New ways of interpreting expressions are designed: the explain pattern makes it easy to "evaluate" new expressions. For example, you can define a new operation on an expression class to support graceful printing or type checking of expressions. If you often create new ways t o interpret expressions, consider using the VIS I to R pattern to avoid modifying these classes that represent grammar.

Related patterns related to interpreter patterns:

Composite pattern: an instance of a compound pattern can be considered as an abstract syntax tree.

Flyweight pattern: explains how to share terminators in the abstract syntax tree.

Iterator: the interpreter can traverse the structure with an iterator.

Visitor: it can be used to safeguard the action of each panel point in the abstract syntax tree in a class.

5. Summary

Through the realization of object-oriented mathematical expression by using Interpreter pattern, I have a comprehensive understanding of Interpreter pattern and cultivated my own object-oriented design idea. In the future, when encountering a specific problem, we can use the method of design pattern to solve it. We can select the appropriate design pattern, or use the combination of design patterns to make the design task complete efficiently and quickly, which will be of great help to our future work and learning.

Reference

[1] Erich gamma Richard helm. Design patterns (the basis of reusable object-oriented software). Trans. Ma Xiaoxing et al. Beijing: China Machine Press, 2019

[2] Geng Xiangyi, Zhang Yueping. Object oriented and design pattern. Tsinghua University press, 2013

[3] Zhong Maosheng, Wang Mingming. Software design mode and its use. Computer application, 2002

[4] Jiahua Li, Tao Wang，Zhengshi Chen，Guoqiang Luo. Machine Learning Algorithm Generated Sales Prediction for Inventory Optimization in Cross-border E-Commerce. International Journal of Frontiers in Engineering Technology (2019), Vol. 1, Issue 1: 62-74.