Interpreter pattern

From Wikipedia, the free encyclopedia

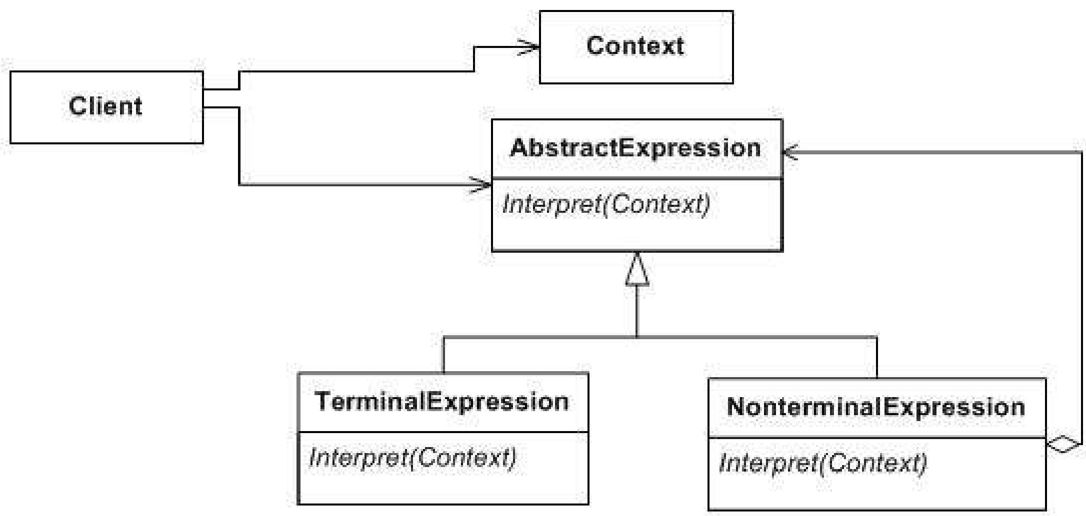
In computer programming, the **interpreter pattern** is a design pattern that specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence for a client.[1]:243 See also Composite pattern.

Contents

Uses

- Specialized database query languages such as SQL.
- Specialized computer languages which are often used to describe communication protocols.
- Most general-purpose computer languages actually incorporate several specialized languages.

Structure



Example

BNF

The following Backus-Naur Form example illustrates the interpreter pattern. The grammar

```
expression ::= plus | minus | variable | number
plus ::= expression expression '+'
minus ::= expression expression '-'
variable  ::= 'a' | 'b' | 'c' | ... | 'z'
digit = '0' | '1' | ... | '9'
number ::= digit | digit number
```

defines a language which contains Reverse Polish Notation expressions like:

```
a b +
a b c + -
a b + c a - -
```

C#

This structural code demonstrates the Interpreter patterns, which using a defined grammar, provides the interpreter that processes parsed statements.

```
namespace DesignPatterns.Interpreter
{
    // "Context"
    class Context
    {
    }

    // "AbstractExpression"
    abstract class AbstractExpression
    {
        public abstract void Interpret(Context context);
    }

    // "TerminalExpression"
    class TerminalExpression : AbstractExpression
    {
        public override void Interpret(Context context)
        {
            Console.WriteLine("Called Terminal.Interpret()");
        }
    }

    // "NonterminalExpression"
    class NonterminalExpression : AbstractExpression
    {
        public override void Interpret(Context context)
        {
            Console.WriteLine("Called Nonterminal.Interpret()");
        }
    }

    class MainApp
    {
        static void Main()
        {
            Context context = new Context();

            // Usually a tree
            ArrayList list = new ArrayList();

            // Populate 'abstract syntax tree'
            list.Add(new TerminalExpression());
            list.Add(new NonterminalExpression());
            list.Add(new TerminalExpression());
            list.Add(new TerminalExpression());

            // Interpret
            foreach (AbstractExpression exp in list)
            {
```

```
                exp.Interpret(context);
            }

            // Wait for user
            Console.Read();
        }
    }
}
```

Java

Following the interpreter pattern there is a class for each grammar rule.

```
import java.util.Map

interface Expression {
    public int interpret(Map<String,Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(int number)       { this.number = number; }
    public int interpret(Map<String,Expression> variables)  { return number; }
}

class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(Map<String,Expression> variables)  {
        return leftOperand.interpret(variables) + rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(Map<String,Expression> variables)  {
        return leftOperand.interpret(variables) - rightOperand.interpret(variables);
    }
}

class Variable implements Expression {
    private String name;
    public Variable(String name)       { this.name = name; }
    public int interpret(Map<String,Expression> variables)  {
        if(null==variables.get(name)) return 0; //Either return new Number(0).
        return variables.get(name).interpret(variables);
    }
}
```

While the interpreter pattern does not address parsing[1]:247 a parser is provided for completeness.

```
import java.util.Map;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;
```

```
    public Evaluator(String expression) {
        Stack<Expression> expressionStack = new Stack<Expression>();
        for (String token : expression.split(" ")) {
            if  (token.equals("+")) {
                Expression subExpression = new Plus(expressionStack.pop(), expressionStack.pop());
                expressionStack.push( subExpression );
            }
            else if (token.equals("-")) {
                // it's necessary remove first the right operand from the stack
                Expression right = expressionStack.pop();
                // ..and after the left one
                Expression left = expressionStack.pop();
                Expression subExpression = new Minus(left, right);
                expressionStack.push( subExpression );
            }
            else
                expressionStack.push( new Variable(token) );
        }
        syntaxTree = expressionStack.pop();
    }

    public int interpret(Map<String,Expression> context) {
        return syntaxTree.interpret(context);
    }
}
```

Finally evaluating the expression "w x z - +" with w = 5, x = 10, and z = 42.

```
import java.util.Map;
import java.util.HashMap;

public class InterpreterExample {
    public static void main(String[] args) {
        String expression = "w x z - +";
        Evaluator sentence = new Evaluator(expression);
        Map<String,Expression> variables = new HashMap<String,Expression>();
        variables.put("w", new Number(5));
        variables.put("x", new Number(10));
        variables.put("z", new Number(42));
        int result = sentence.interpret(variables);
        System.out.println(result);
    }
}
```

See also

- Interpreter (computing)
- Backus-Naur form
- Domain-specific language
- *Design Patterns*
- Combinator#Combinatory logic in computing

References

1. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

External links

- Interpreter implementation (http://lukaszwrobel.pl/blog/interpreter-design-pattern) in Ruby
- Interpreter implementation (https://github.com/jamesdhutton/Inte

The Wikibook *Computer Science Design Patterns* has a page on the topic of:

rpreter) in C++
- SourceMaking tutorial (http://sourcemaking.com/design_patterns/interpreter)
- Interpreter pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?InterpreterPattern)

Interpreterimplementations in various languages

Retrieved from "https://en.wikipedia.org/w/index.php?title=Interpreter_pattern&oldid=726785576"

Categories: Software design patterns
