WIKIPEDIA

Interpreter pattern

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

Overview

The Interpreter [2] design pattern is one of the twenty-three well-known *GoF design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Interpreter design pattern solve? [3]

- A grammar for a simple language should be defined
- so that sentences in the language can be interpreted.

When a problem occurs very often, it could be considered to represent it as a sentence in a simple language (Domain Specific Languages) so that an interpreter can solve the problem by interpreting the sentence.

For example, when many different or complex search expressions must be specified. Implementing (hard-wiring) them directly into a class is inflexible because it commits the class to particular expressions and makes it impossible to specify new expressions or change existing ones independently from (without having to change) the class.

What solution does the Interpreter design pattern describe?

- Define a grammar for a simple language by defining an `Expression` class hierarchy and implementing an `interpret()` operation.
- Represent a sentence in the language by an abstract syntax tree (AST) made up of `Expression` instances.
- Interpret a sentence by calling `interpret()` on the AST.

The expression objects are composed recursively into a composite/tree structure that is called *abstract syntax tree* (see Composite pattern).
The Interpreter pattern doesn't describe how to build an abstract syntax tree. This can be done either manually by a client or automatically by a parser.

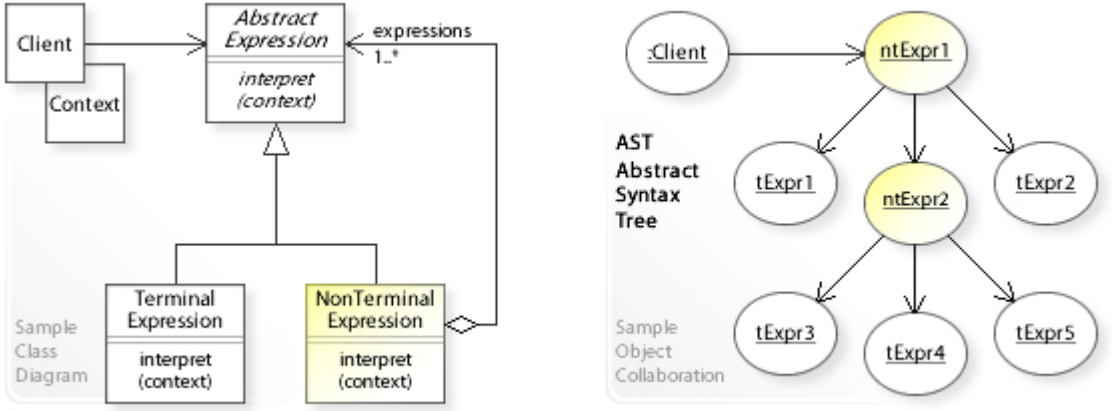
See also the UML class and object diagram below.

Uses

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

Structure

UML class and object diagram

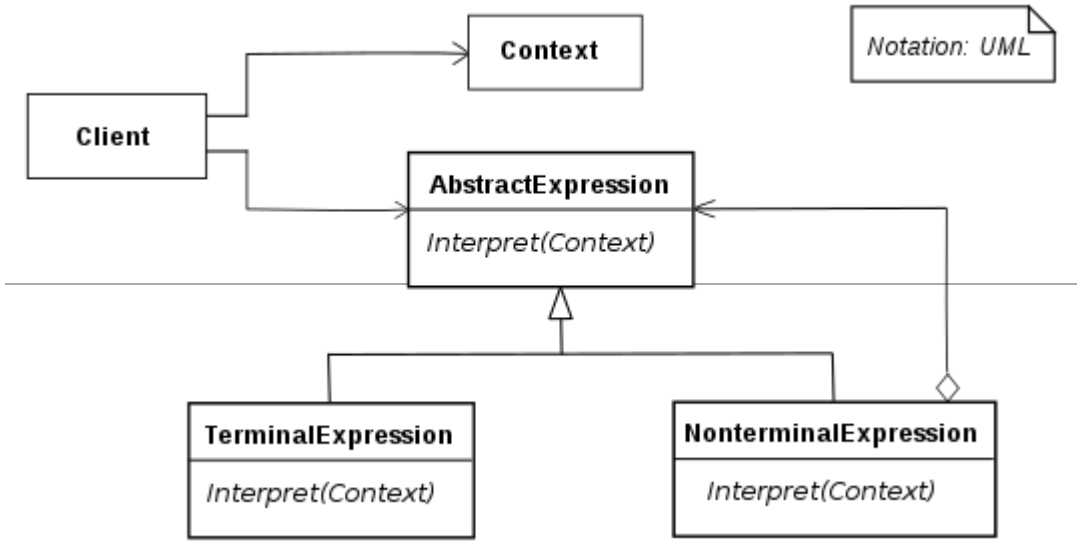


A sample UML class and object diagram for the Interpreter design pattern. [4]

In the above UML class diagram, the `Client` class refers to the common `AbstractExpression` interface for interpreting an expression `interpret(context)`.
The `TerminalExpression` class has no children and interprets an expression directly.
The `NonTerminalExpression` class maintains a container of child expressions (`expressions`) and forwards interpret requests to these `expressions`.

The object collaboration diagram shows the run-time interactions: The `Client` object sends an interpret request to the abstract syntax tree. The request is forwarded to (performed on) all objects downwards the tree structure.
The `NonTerminalExpression` objects (`ntExpr1,ntExpr2`) forward the request to their child expressions.
The `TerminalExpression` objects (`tExpr1,tExpr2,…`) perform the interpretation directly.

UML class diagram



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 that 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()
        {
            var context = new Context();

            // Usually a tree
            var list = new List<AbstractExpression>();

            // 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);
            }
        }
    }
}
```

Java

Following the interpreter pattern, we need to implement the interface Expr with a lambda (it can be a class) for each grammar rule.

```
public class Interpreter {
  @FunctionalInterface
  public interface Expr {
    int interpret(Map<String, Integer> context);

    static Expr number(int number) {
      return context -> number;
    }

    static Expr plus(Expr left, Expr right) {
      return context -> left.interpret(context) + right.interpret(context);
    }

    static Expr minus(Expr left, Expr right) {
      return context -> left.interpret(context) - right.interpret(context);
    }

    static Expr variable(String name) {
      return context -> context.getOrDefault(name, 0);
    }
  }
...
```

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

```
...
  private static Expr parseToken(String token, ArrayDeque<Expr> stack) {
    Expr left, right;
    switch(token) {
    case "+":
      // it's necessary to remove first the right operand from the stack
      right = stack.pop();
      // ..and then the left one
      left = stack.pop();
      return Expr.plus(left, right);
    case "-":
      right = stack.pop();
      left = stack.pop();
      return Expr.minus(left, right);
    default:
      return Expr.variable(token);
    }
  }
  public static Expr parse(String expression) {
    var stack = new ArrayDeque<Expr>();
    for (var token : expression.split(" ")) {
      stack.push(parseToken(token, stack));
    }
```

```
    return stack.pop();
  }
...
```

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

```
...
  public static void main(final String[] args) {
    var expr = parse("w x z - +");
    var context = Map.of("w", 5, "x", 10, "z", 42);
    var result = expr.interpret(context);
    System.out.println(result);    // -27
  }
}
```

See also

- Backus–Naur form
- Combinatory logic in computing
- *Design Patterns*
- Domain-specific language
- Interpreter (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.
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 243ff. ISBN 0-201-63361-2.
3. "The Interpreter design pattern - Problem, Solution, and Applicability" (http://w3sdesign.com/?gr=b03&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
4. "The Interpreter design pattern - Structure and Collaboration" (http://w3sdesign.com/?gr=b03&ugr=struct). *w3sDesign.com*. Retrieved 2017-08-12.

External links

- Interpreter implementation (http://lukaszwrobel.pl/blog/interpreter-design-pattern) in Ruby
- Interpreter implementation (https://github.com/jamesdhutton/Interpreter) in C++
- SourceMaking tutorial (http://sourcemaking.com/design_patterns/interpreter)
- Interpreter pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?InterpreterPattern)