Interpreter [GoF]

Intent

Given a language, define a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Motivation

Many problems can be expressed as simple formal languages; instances of the problem then are sentences of that language. Examples:

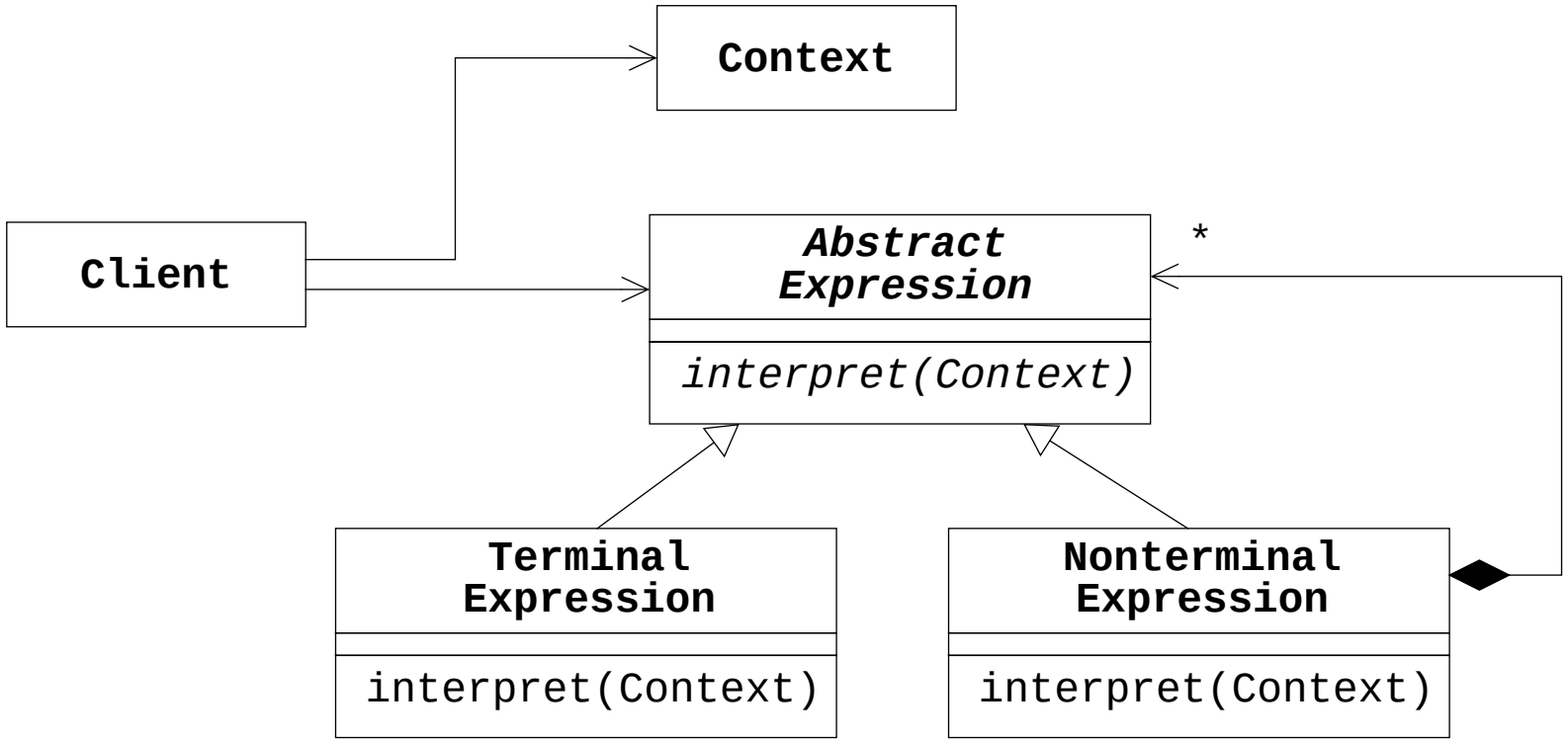
- Terms and expressions such as `(a+b)*3` denoting values.
- Query terms to search in an index.
- Regular expressions.

Applicability

Use the interpreter pattern if:

- you have a simple language to interpret;
- you can represent sentences in the language as abstract syntax trees (AST).

Structure


Context
Client
Abstract Expression
interpret(Context)
*
Terminal Expression
interpret(Context)
Nonterminal Expression
interpret(Context)

Participants

- **`AbstractExpression`**:
  - declares an abstract `interpret` method that is common to all node in the abstract syntax tree
- **`TerminalExpression`**:
  - implements the `interpret` method associated with terminal symbols of the grammar
  - a concrete class is required for each terminal symbol of the grammar
- **`NonterminalExpression`**:
  - one of such class is required for each rule $R ::= R_1\ R_2\ \ldots\ R_n$
  - maintains instance variables of type `AbstractExpression` for each of the symbols $R_1\ R_2\ \ldots\ R_n$
  - implements the concrete `interpret` method. Method `interpret` is applied iteratively on the instance variables representing $R_1\ R_2\ \ldots\ R_n$
- **`Context`**:
  - contains information that is global to the interpreter, for example, the actual values of variables
- **`Client`**:
  - builds (or is given) an abstract syntax tree that is assembled from instances of `NonterminalExpression` and `TerminalExpression`

- invokes the `interpret` method

Collaborations

- The client builds the abstract syntax tree. Then the client initializes the `Context` of the interpreter and invokes the `interpret` method.
- The `interpret` methods at each (non-) terminal node use the context to store and access the state of the interpreter.

Consequences

- It is relatively easy to change and extend the grammar.
- Complex grammars are difficult to maintain:
  Use parser generators that allow to specify the concrete grammar as well the abstract syntax tree, and that generate all depending classes. For examples: `javacc` and `jjtree`, or `antlar`.
- Adding new interpretations:
  The interpreter pattern makes it easy to interpret the abstract syntax tree in a new way by defining new methods on the grammar classes denoting the nodes of the AST. You might consider using the Visitor pattern to avoid changing the grammar classes.

Implementation

Interpreter and Composite share many implementation issues. Specific to Interpreter are:

- Interpreter does not tell how to create the AST. You might use the Little Grammar pattern, or parser generators.
- Defining the `interpret` methods using Visitor.
- Sharing terminals using Flyweight.

Sample Code

An example for evaluating expression on integers is given. The grammar is:[1]

```
// BNF.
BooleanExp  ::=   Constant | Variable | OrExp | AndExp | NotExp
AndExp      ::=   BooleanExp '*' BooleanExp
OrExp       ::=   BooleanExp '+' BooleanExp
NotExp      ::=   '~' BooleanExp
Variable    ::=   // any identifier according to: [A-z][A-z]*
Constant    ::=   'true' | 'false'
```

1. Operator precedence and the use of parenthesis for grouping ignored.

We define the method `interpret` on the AST which evaluates an expression in a context that assigns `true` or `false` to each variable. The details of the `Constant`, `Variable`, and `AndExpr` classes are given only.

Common base class for all kind of expression classes:

```
public abstract class BooleanExp {
   public abstract boolean interpret(Context ctx);
}
```

Class `Context` defines the mapping from variables to `boolean` values:

```
public class Context {
   private HashMap map = new HashMap();
   public boolean lookup(Variable var) { ... }
   public void assign(Variable var, boolean value) { ... }
}
```

Class `Constant` represents a `boolean` constant:

```
public class Constant extends BooleanExp {
   private boolean value;
   public Constant(boolean value) {
      this.value = value;
   }
   public boolean interpret(Context ctx) {
      return value;
   }
}
```

Class `Variable` represents a named variable:

```
public class Variable extends BooleanExp {
   private String name;
   public Variable(String name) {
      this.name = name;
   }
   public boolean interpret(Context ctx) {
      return ctx.lookup(this);
   }
}
```

An object of class `AndExp` represents an expression made by “anding” two `BooleanExp` instances:

```
public class AndExp extends BooleanExp {
   private BooleanExp left, right;

   public AndExp(BooleanExp left, BooleanExp right) {
      this.left = left; this.right = right;
   }

   public boolean interpret(Context ctx) {
      return left.interpret(ctx) && right.interpret(ctx);
   }
}
```

The code to interpret the expression `(true * x) + (y * (~ x))` is:

```
// E.g. in a main():
   BooleanExp exp;
   Context ctx = new Context();

   Variable x = new Variable("x");
   Variable y = new Variable("y");
   exp = new OrExp(
            new AndExp(new Constant(true), x),
            new AndExp(y, new NotExp(x))
            );
   ctx.assign(x, false);
   ctx.assign(y, true);

   boolean result = exp.interpret(ctx);
```

Related Patterns

- Composite for the AST.
- Visitor to move the operations from the nodes of the AST into one class.
- Flyweight for sharing terminal symbols.
- Iterator for traversing the nodes of non-terminals.