InterBit
Training & Consulting Ltd.

Interpreter Pattern

Chapter Content

- Interpreter Pattern Overview
- Interpreter UML Diagram
- Interpreter Pattern Example

Interpreter Pattern Overview

- Allows the programmer to:
  - Define a language.
  - Given a valid expression in this language, parse it and create a matching tree.
- Caution:
  - parsers are usually more sophisticated and less memory-consuming.
  - Interpreter is enough when:
    - Language is simple.
    - Efficiency is not crucial.

Motivation

- The Interpreter is one of the Design Patterns published in the GoF book which is not really used
- Usually the Interpreter Pattern is described in terms of formal grammars, like it was described in the original form in the GoF book
- But the area where this design pattern can be applied can be extended.

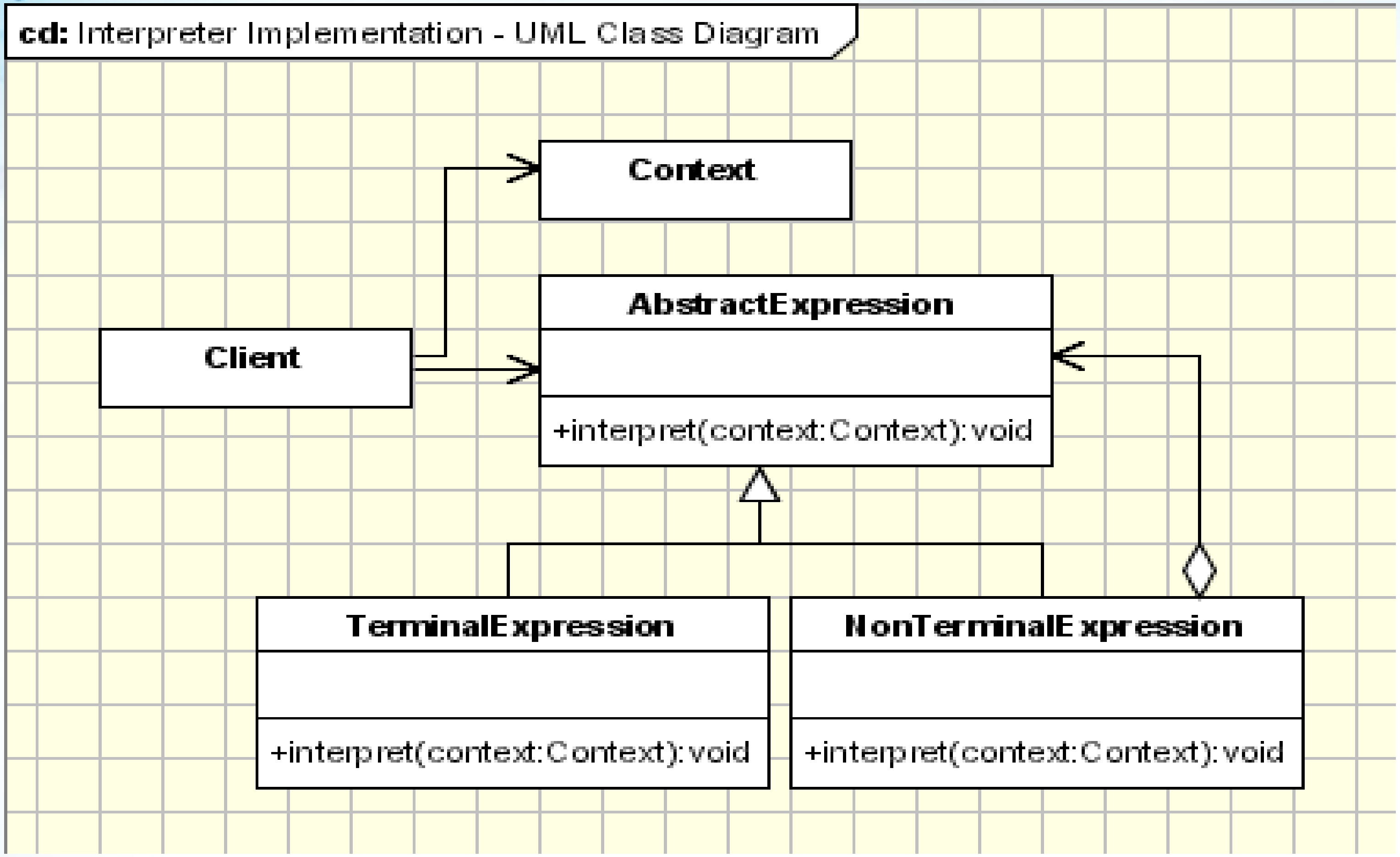
Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

Implementation

- The implementation of the Interpreter pattern is just the use of the composite pattern applied to represent a grammar.
- The Interpreter defines the behavior while the composite defines only the structure.

UML Diagram


cd: Interpreter Implementation - UML Class Diagram
Context
AbstractExpression
+interpret(context:Context):void
Client
TerminalExpression
+interpret(context:Context):void
NonTerminalExpression
+interpret(context:Context):void

Interpreter suggests modeling the domain with a recursive grammar. Each expression in the grammar is either a ‘composite’ (expression that references other expressions) or a terminal (a leaf node in a tree structure). Interpreter relies on the recursive traversal of the Composite pattern to interpret the ‘expressions’ it is asked to process.

Roman Numerals Converter

- The classical example of the interpreter pattern is the one of interpreting the roman numerals
- The expression to be interpreted is a string which is put in the context
- The context consists of the remaining unparsed Roman Numeral string and of the result of the numeral that are already parsed
- The context is passed to one of four sub-interpreters based on the type of interpreting (Thousand, Hundred, Ten, One).
- In this example we're using only ***TerminalExpression***.

Roman Numerals Converter

- The following participant classes are involved in this example:
- **Context** - keeps the current string that has to be parsed and the decimal that contains the conversion already done
    - Initially the context keeps the full string that has to be converted and 0 for the output decimal.

Roman Numerals Converter

- **Expression** - Consists of the interpret method which receives the context
- Based on the current object it uses specific values for Thousand, Hundred, Ten, One and a specific multiplier
- **ThousandExpression**, **HundredExpression**, **TenExpression**, **OneExpression** (TerminalExpression) - are used to define each specific expression
  - Usually, the TerminalExpression classes implements the interpret method
  - In our case the method is already defined in the base Expression class and each TerminalExpression class defines its behavior by implementing the abstract methods: *`one(), four(), five(), nine(), multiplier()`*.
  - It is a template method pattern.

Roman Numerals Converter

- Main(Client) - In our little example this class is responsible for building the syntax tree representing a specific sentence in the language defined by the grammar.
- After the syntax tree is built, the main method is invoking the `interpret()` method.

Roman Numerals Converter

The Context (part 1)

```
public class Context {
    private String input;
    private int output;
    public Context(String input) {
      this.input = input;
    }
    public String getInput(){
      return input;
    }
    public void setInput(String input) {
      this.input = input;
    }
    public int getOutput() {
      return output;
    }
    public void setOutput(int output) {
      this.output = output;
    }
}
```

Roman Numerals Converter

InterBit
Training & Consulting Ltd.

The Expression Hierarchy (part 2)

```
public abstract class Expression {
    public void interpret(Context context) {
      if (context.getInput().length() == 0)
        return;
      if (context.getInput().startsWith(nine())) {
        context.setOutput(context.getOutput() + (9 * multiplier()));
        context.setInput(context.getInput().substring(2));
      } else if (context.getInput().startsWith(four())) {
        context.setOutput(context.getOutput() + (4 * multiplier()));
        context.setInput(context.getInput().substring(2));
      } else if (context.getInput().startsWith(five())) {
        context.setOutput(context.getOutput() + (5 * multiplier()));
        context.setInput( context.getInput().substring(1));
      }
      while (context.getInput().startsWith(one())) {
        context.setOutput(context.getOutput() + (1 * multiplier()));
        context.setInput(context.getInput().substring(1));
      }
    } // continued next slide
```

Roman Numerals Converter

The Expression Hierarchy (part 3)

```
    // continued
    public abstract String one();
    public abstract String four();
    public abstract String five();
    public abstract String nine();
    public abstract int multiplier();
}

public class ThousandExpression  extends Expression{
    public String one() { return "M"; }
    public String four(){ return " "; }
    public String five(){ return " "; }
    public String nine(){ return " "; }
    public int multiplier() { return 1000; }
}
```

Roman Numerals Converter

The Expression Hierarchy (part 4)

```
public class HundredExpression extends Expression{
    public  String one() { return "C"; }
    public  String four(){ return "CD"; }
    public  String five(){ return "D"; }
    public  String nine(){ return "CM"; }
    public  int multiplier() { return 100; }
}

public class TenExpression  extends Expression{
    public String one() { return "X"; }
    public String four(){ return "XL"; }
    public String five(){ return "L"; }
    public String nine(){ return "XC"; }
    public int multiplier() { return 10; }
}
```

Roman Numerals Converter

The Expression Hierarchy (part 5)

```
public class OneExpression  extends Expression{
    public String one() { return "I"; }
    public String four(){ return "IV"; }
    public String five(){ return "V"; }
    public String nine(){ return "IX"; }
    public int multiplier() { return 1; }
}
```

Roman Numerals Converter

The Client (part 6)

```
public class MainInterpreter {


      public static void main(String[] args) {
            String roman = "MCMXXVIII";
            Context context = new Context(roman);
            // Build the 'parse tree'
            ArrayList<Expression> tree = new ArrayList<Expression>();
            tree.add(new ThousandExpression());
            tree.add(new HundredExpression());
            tree.add(new TenExpression());
            tree.add(new OneExpression());
            // Interpret
            for (Iterator it = tree.iterator(); it.hasNext();) {
              Expression exp = (Expression)it.next();
              exp.interpret(context);
            }
            System.out.println(roman + " = " + Integer.toString(context.getOutput()));
      }
}
```

Interpreter Pattern Conclusion

- The Interpreter pattern has a limited area where it can be applied
- We can discuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions and this is the reason why this pattern is not so frequently used
- This pattern can be applied for parsing light expressions defined in simple grammars and sometimes in simple rule engines