Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 23: Interpreter Design Pattern

Contents

23.1 Introduction

The interpreter design pattern is a behavioral pattern relying on its inheritance structure to achieve its purpose. It represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes. When the interpreter is executed on a composite node in the hierarchy it propagates to all the descendants of such node.

In this lecture we assume a basic knowledge of grammars. We illustrate by example how the pattern is applied to translate given grammars into implementations.

23.2 Interpreter Design Pattern

23.2.1 Identification

Name	**Classification**	**Strategy**
Interpreter	Behavioural	Inheritance
Intent		
Given a language, define a represention for its grammar along with aninterpreter that uses the representation to interpret sentences in thelanguage. ([2]:243)		

23.2.2 Structure

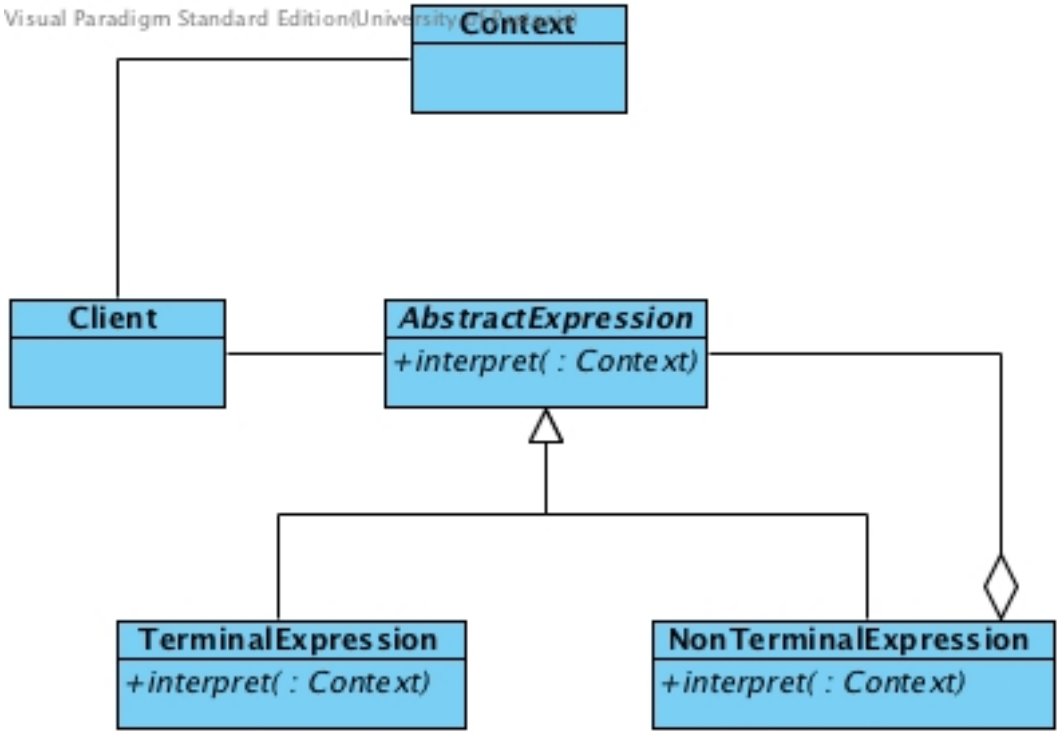

Figure 1: The structure of the Interpreter Design Pattern

23.2.3 Participants

AbstractExpression

- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression

- implements an Interpret operation associated with terminal symbols in the grammar.
- an instance is required for every terminal symbol in a sentence.

NonterminalExpression

- one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
- maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
- implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Context

- contains information that's global to the interpreter.

Client

- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
- invokes the Interpret operation.

23.2.4 Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a language that can be expressed in terms of a formal grammar, then problems could be easily solved with an interpretation engine representing the grammar. [3].

23.3 Interpreter pattern explained

23.3.1 Improvements achieved

Improved adaptability

It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.

Implementation can be automated

Once a grammar is defined, the class design and its implementation is completely determined by the rules of the grammar. The generation of code for classes defining nodes in the abstract syntax tree can often be automated with a compiler or parser generator using the grammar as input.

23.3.2 Situations where other solutions may be more suitable

The pattern is not suitable for complex grammars. If the grammar is complex the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.

If you need multiple *interpreter* operations, then it might be better to use the Visitor pattern. Here every *interpret* operation can be implemented in a separate *visitor* object. For example, a grammar for a programming language will have many operations on abstract syntax trees, such as type-checking, optimization, code generation, and so on. It will be more likely to use a visitor to avoid defining these operations on every grammar class.

23.3.3 Common Misconception

The pattern require the existence of an abstract syntax tree of the expression that has to be interpreted. It is often assumed that the compilation of the abstract syntax tree of the sentence that needs to be interpreted is part of the pattern and that the pattern can only be implemented if a parser of text input is involved. However, the Interpreter pattern doesn't explain how to create an abstract syntax tree.

The pattern does not address parsing. The abstract syntax tree can be created by a table-driven parser, by a hand-crafted (usually recursive descent) parser, or directly by the client. Thus the abstract syntax tree is generated from input provided through a GUI or a series of prompts. For example the syntax tree of a mathematical expression may as well be compiled using the following code assuming that the methods that are called in the switch statement will also prompt the user to enter appropriate information.

```
Expression* inputExpression(string prompt)
{
        Expression* exp;
        cout << "Specify the type of the " << prompt << endl;
        cout << "1. A value" << endl;
        cout << "2. A variable" << endl;
        cout << "3. An operation" << endl;
        cout << "Enter your choice: ";
        int choice;
        cin >> choice;
        switch (choice)
        {
                case 1: exp = inputValue(); break;
                case 2: exp = inputVariable(); break;
                case 3: exp = inputOperation(); break;
```

```
        }
        return exp;
}
```

In this case the `inputOperation()` method will typically prompt the user for an operation and then recursively call this method for the left- and right operands of the operation.

23.3.4 Related Patterns

Template Method
: The template method and the interpreter design patterns are the only two Gamma:1995 behavioural patterns that uses inheritance as its basic strategy to achieve its purpose. All other Gamma:1995 behavioural patterns uses delegation.

Composite
: Interpreter is an application of the Composite pattern. It ads specific behaviour to the composite structure namely to interpret the elements of the composite structure. Note that is more specific than just an operation distributed over a class hierarchy that uses the Composite pattern. It is specifically aimed at dealing with problems that are specified in terms of grammars.

Flyweight
: Both Flyweight and Interpreter share symbols. Interpreter share terminal symbols within the abstract syntax tree.

Visitor
: Both Interpreter and Visitor adds behaviour to a composite structure. Where interpreter adds a simplistic behaviour, visitor allows for more generic and adaptable behaviour. The similarity between these patterns is deep. The operations of an interpreter can always be refactored into *interpreter* visitors.

State
: The interpreter design pattern and the state design pattern are different ways to solve interpretation. Where the interpreter pattern provide a way to interpret parse trees directly while it is possible to first transform a parse three into a state machine and then applying the state pattern to solve the same problem.

23.4 How to implement the interpreter pattern

The implementation of the interpreter pattern is useful to achieve one of the following:

- to verify if a given sentence complies with the grammar
- to generate a sentence that complies with a grammar
- to determine a value of a sentence that complies with a grammar

In this section we explain the process of implementing this pattern at the hand of an example that verifies if a given sentence complies with a language. In Section 23.5 we present an example that applies this pattern to generate a sentence that complies with a grammar. Lastly, in Section 23.6, we present a tutorial that guides the reader through the process to implement the pattern to calculate the value of a mathematical expression.

23.4.1 Define a grammar

The interpreter pattern is only applicable if the solution to the problem at hand can be expressed in terms of a formal grammar. If the problem offers a justifiable return on investment one can define a grammar and then build an interpretation engine using this pattern to process the solutions.

23.4.2 Use the grammar to design the system

The Interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes. Thus the grammar rule `alternation ::= expression '|' expression`, should be represented in the design with the classes shown in Figure 2.

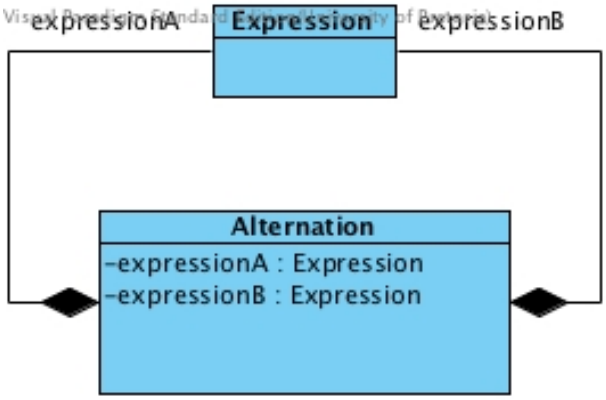

Figure 2: Classes representing `alternation ::= expression '|' expression`

23.4.3 Implement the design using the grammar

The constructor of each class should instantiate its instance variables. This is typically done by implementing only one constructor that requires parameters to specify the values for all its instance variables. The following is an example of the constructor of this `Alternation` class assuming the `Expression` class implements an assignment operator.

```
Alternation::Alternation(Expression a, Expression b)
{
    expressionA = a;
    expressionB = b;
}
```

The `interpret()` method of each class should be implemented. For example the `interpret()` method of an application that verifies if a sentence complies will return a boolean value. It will return true if the input matches, and will return false if it does not. Assume the the `Alternation` class given above must check if the input matches any of its alternatives. Below is the implementation of this method. Note how the interpretation is delegated to its instance variables.

```
bool Alternation :: interpret (string input){
    bool a = expressionA.interpret(input);
    bool b = expressionB.interpret(input);
    return a || b;
}
```

23.5 Example

The interpreter pattern is the translation of a grammar to an implementation. Usually grammars are associated with parsing and interpretation of parsed text. Most examples illustrating the interpreter pattern includes a parser and illustrates the pattern at the hand of grammars that identify expressions that are parsed before they are interpreted. It is, however, clearly stated in [2] that the pattern does not include parsing.

23.5.1 The problem

For our example we chose an example given by Huston [3]. It does not include the parsing of expressions. We deem it a suitable example to illustrate the design of a grammar and the translation of such grammar to an implementation. It is an expression of the well known Towers of Hanoi puzzle in terms of a grammar that is used to implement a solution of the problem.

The Towers of Hanoi puzzle consists of three pegs and a number disks with different sizes. The goal is to reposition the stack of disks from one peg to another peg by moving one disk at a time, and, never placing a larger disk on top of a smaller disk. An interactive example to solve the puzzle is available at [1]. This is a classic problem for teaching recursion in data structures courses.

Here a language is designed that characterizes this problem domain. The language is then mapped to a grammar, and the grammar is implemented with an interpretation engine that apples the interpreter patten. This application can now be applied to solve the puzzle. The implementation writes the solution in terms of the moves required to achieve the required end state.

23.5.2 A language

The language required to describe a solution can be defined in terms of moves. A simple move is described in terms of the source peg and the destination peg. Executing a move described as `A B`, means the top disk on peg A is moved to peg B. It will be valid if peg B is empty or the top disk on peg B is larger than the top disk on peg A.

A complex move is defined as the combination of moves needed to move a stack of n disks from one peg to another. Executing this move can be described as `n A B`. Such move is valid if the bottom disk of top n disks on peg A is smaller than the top disk on peg B.

Every solution to move n disks from one peg to a specified peg can be now described in terms of a complex move, followed by a simple move, followed by a complex move. For example the complex move `4 A C` can be refined as `3 A B` followed by `A C` followed by `3 B C`. The logic is moving the top 3 disks out of the way, move the bottom of the four disks to its required destination, and thereafter move the top three disks to the required destination.

23.5.3 A grammar

In the above description of the solution to the general problem we identified types of moves and a production rule. Assuming the pegs are labelled A, B and C, this can be formulated in the following grammar to define the language to describe the solution:

```
1  Move ::= complexMove | simpleMove
2  simpleMove ::= peg1 peg2
3  complexMove ::= size peg1 peg2
4  size peg1 peg2 ::= size2 peg1 peg3, peg1 peg2, size2 peg3 peg2
5  1 peg1 peg2 ::= peg1 peg2
6  peg1 ::= A | B | C
7  peg2 ::= (A | B | C) and !(peg1)
8  peg3 ::= (A | B | C) and !(peg1) and !(peg2)
9  size ::= (an integer)
10 size2 ::= (value of (size - 1))
```

Rules 1, 2 and 3 are general rules while rules 6 to 10 are specific rules stating the allowable values and relationships between terminal symbols. Rules 4 and 5 are production rules specifying how a complex move can be expressed in terms of other moves. It includes detail about the values and relationships between the terminal symbols comprising the expression. The following expressions are a generalisations of these rules:

```
4  complexMove ::= complexMove, simpleMove, complexMove
5  complexMove ::= simpleMove
```

23.5.4 Mapping the grammar to a design

When mapping the grammar to a design. One look at the general rules and their relation to one another. You will notice that the classes in the class diagram in Figure 3 correspond with the named items (Move, simpleMove and complexMove) in the grammar defined in Section 23.5.3. The other items like the peg names and the values in the variable called `size` in the given grammar feature as instance variables of these classes.

The design represents the abstract rules defined in the given grammar. These are the abstract versions of rules 1 to 5. The application of other rules specifying the detail about peg names and values and their relation to one another will be observed in the implementation.

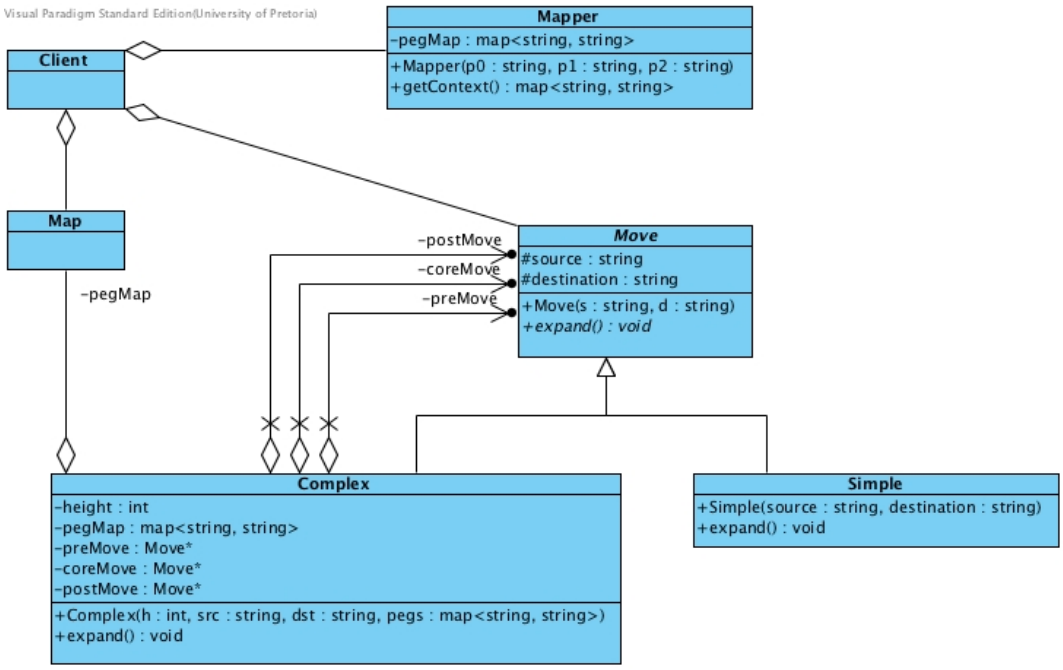

Figure 3: Class design of the grammar in section 23.5.3

The following table identifies of these classes as participants of the interpreter design pattern:

Participant	**Entity in application**
Abstract Expression	Move
Terminal Expression	Simple
Non-Terminal Expression	Complex
Context	map
Client	Client
interpret()	expand()

23.5.5 Implementing the design

Client

- This client prompts the user to specify names for the source peg, the destination peg and an auxiliary peg. It also prompts the user for the number of disks that is to be moved from the specified source peg to the required destination peg.
- The input data is passed to the `Mapper` class to compile the context (see later).
- The client is now able to produce the solution to the puzzle for the given context in a single call to the `expand()` method of the complex move to move the entire stack of disks from the source disk to the destination disk.

Mapper

- The `Mapper` class seems as if it should act as the context. However, in this implementation it is merely a helper class to construct the context.

map

- In this implementation the map serves as context. It is a lookup table that specifies the name of the spare peg when the source and destination pegs are known.
- We deviate from the design of the interpreter pattern in how we use the context. The design of the interpreter pattern suggests that the context be passed as parameter to the `interpret()` method. However, in this implementation this context is also needed by the constructor of the `Complex` class. It is also not used by the `Simple` class. It was therefore decided to pass it to the constructor of the `Complex` class and keep it as an instance variable of a `Complex` object.

Move

- This is the interface to all possible moves.
- It defines the virtual method `expand()` that interprets the specific move.
- Since all moves in the puzzle specifies a source peg and a destination peg. It was decided to locate these as instance variables of a move.

Simple

- This class represents a terminal expression. This is because it provides a concrete move of one disk from one peg to another.
- It implements the virtual method `expand()` that interprets the specific move. This method contains a `cout` statement describing the move. In other more typical implementations if the interpreter pattern the `interpret()` method will typically return single result.

Complex

- This is the heart of the interpreter pattern. It is specified that a class should be implemented for each of the production rules in the grammar. In this example rule 4 of the grammar defined in Section 23.5.3, is the only production rule. Therefore, we implement only this one Non-Terminal expression.

- The constructor applies the associated production rule to create its sub-ordinates as instance variables of the class.
- The abstract version of the language rule is reflected in the definition of the instance variables of this class.
- The detailed version of this rule and its consequences are applied to construct these instance variables when an object of this class is instantiated.
- As prescribed in the interpreter pattern, the implementation of the `expand()` method simply calls the `expand()` methods of its instance variables.

The C++ code of this example can be found in the tarball called `L30_Interpreter.tar.gz`.

23.6 Tutorial

We present an example as a tutorial. We only provide partial solutions for the the development steps and leave the completion of these steps to the reader.

23.6.1 The problem

We need to write an application that can evaluate mathematical expressions written in postfix notation. Only binary operations +, - and * with their usual interpretation are supported. The operands for the operations may only be integers or variable names.

23.6.2 A grammar

The language for mathematical expressions is well established. In this context we will allow variable names to be strings of any length consisting of only alphabetical characters. The following rules are part of our grammar:

```
expression ::= sum | difference | product | variable | number
sum ::= expression expression '+'
number ::= int
```

The last rule simply specifies the data type of this terminal symbol. The reader is invited to write rules for the compound expressions `difference` and `product` and also provide a rule for the terminal of type `variable`.

The eager reader can also add rules for the integer operations `quotient` with operator / and `mod` with operator %.

23.6.3 Mapping the grammar to a design

When mapping the grammar to a design, each grammar rule is represented by a class.

There should be a class for each terminal symbol. In this example they are `variable` and `number`. These classes are directly derived from the abstract class `Expression` because

they appear in the right hand side of the first rule indicating that they are kinds of expressions. Note in Figure 4 that the `Number` class representing `number` contains an instance variable of the type as specified by the grammar rule. It also implements a constructor and the `interpret()` method. The reader is invited to add a class representing `variable` to the class diagram in Figure 4.

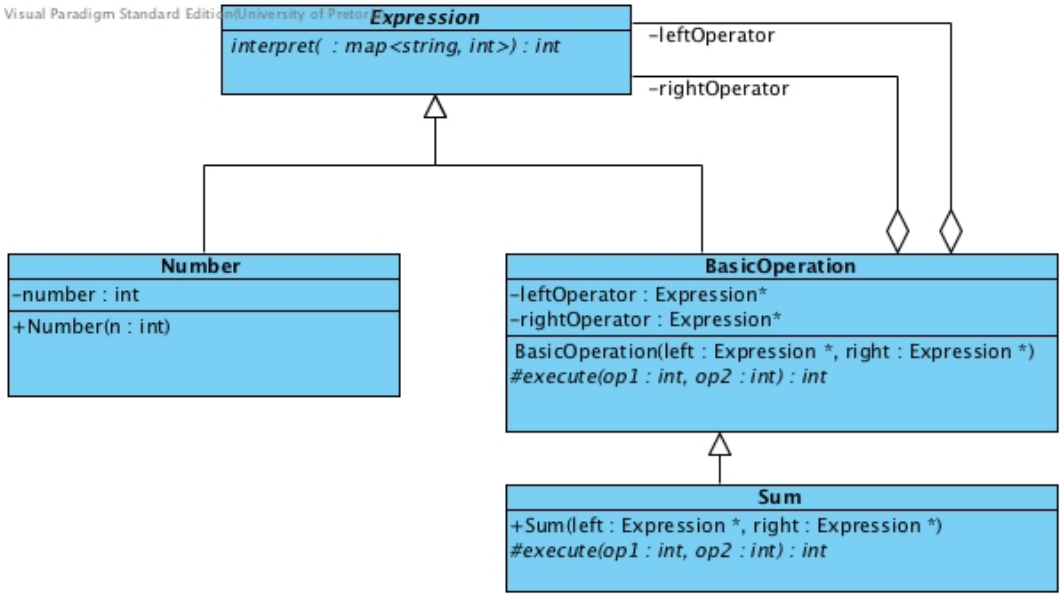

Figure 4: Class design of the grammar in section 23.6.2

There should also be a a class for each production rule. The class `Expression` represents the first rule. Classes that should derive from it are the classes representing `sum`, `difference`, `product`, textttvariable and `number`. In our example we have observed that all the classes representing operations will be similar to the class representing `sum`. Therefore, we have defined an abstract interface for them. They will all have two operands that may be any kind of expression. We have defined these variables in this abstract interface. This abstract interface must, similar to the classes representing terminals, implement a constructor and the `interpret()` method. In this case we have implemented it as a template method that will delegate the core of the operation to its derived class using the virtual `execute()` method.

The reader is invited to add classes for the rules for the compound expressions that he/she added to the grammar.

23.6.4 Implementing the design

When implementing the design one need to have an abstract syntax tree of the expression that needs to be interpreted, compile the context needed for interpretation and inplement the contsructors as well as the interpret method in each of the classes in the design.

Syntax Tree

The compilation of the syntax tree is usually done by implementing a parser. The implementation of a parser or other means of compiling the syntax tree is left as an exercise.

Hint: see Section 23.3.3.

Context

In this example the context entails the values of the variables. It is advised to define the context as a `map<string, int>`. A `pair<string, int>` can be inserted to this map for each variable that is encountered in the expression. This will provide a lookup table that can be used in the implementation the interpret method of the `Variable` class. Since the creation of this context is closely related to the process to build the abstract syntax tree this part of the implementation is left to the reader.

Constructors

Each constructor needs to instantiate its instance variables as specified by the class design. Here we list the constructors of all the classes in Figure 4 that has instance variables. The implementation of the constructors of the rest of the classes are left for the reader to complete.

```
Number::Number(int value): number(value)
{}

BasicOperation::BasicOperation(Expression* left, Expression* right)
{
        leftOperator = left;
        rightOperator = right;
}

Sum::Sum(Expression* left, Expression* right)
     : BasicOperation(left, right)
{}
```

Interpret method

Each class has to implement the `interpret()` method. It is assumed that the context is passed as parameter to the interpret method. Typically the interpret method of terminal symbols simply return its value while composite nodes will recursively call the interpret method of its instance variables. Here we list the implementation of this method of the `Number` and `BasicOpertion` classes in Figure 4 as well as the `execute()` method of the `Sum` class to illustrate how this can be achieved. The implementation of the interpret/execute methods of the rest of the classes are left for the reader to complete.

```
int Number::interpret( map<string, int>)
{
     return number;
}

int BasicOperation::interpret(map<string, int> variables)
{
```

```
    return execute(leftOperator->interpret(variables),
        rightOperator->interpret(variables));
}

int Sum::execute(int left, int right)
{
    return left + right;
}
```

References

[1] Anonymous. Tower of hanoi. `http://www.mathsisfun.com/games/towerofhanoi.html` edited by Rod Pierce. [Online: Accessed 3 October 2011].

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[3] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].