Design Assignment Report

Iván García Fernández & Daniel Quintillán Quintillán

December 21, 2018

Abstract

This design assignment is composed of two sections, *CSV Data Analyzer* and *Arithmetic Expressions*. We have implemented solutions to both exercises by following the design patterns and principles explained below. Following our research, we conclude that CSV Data Analyzers can be accurately represented by state machines, whereas Arithmetic Expressions closely resemble tree-like structures. We have also created UML diagrams as a means of documenting the structure and functionallity of our code.

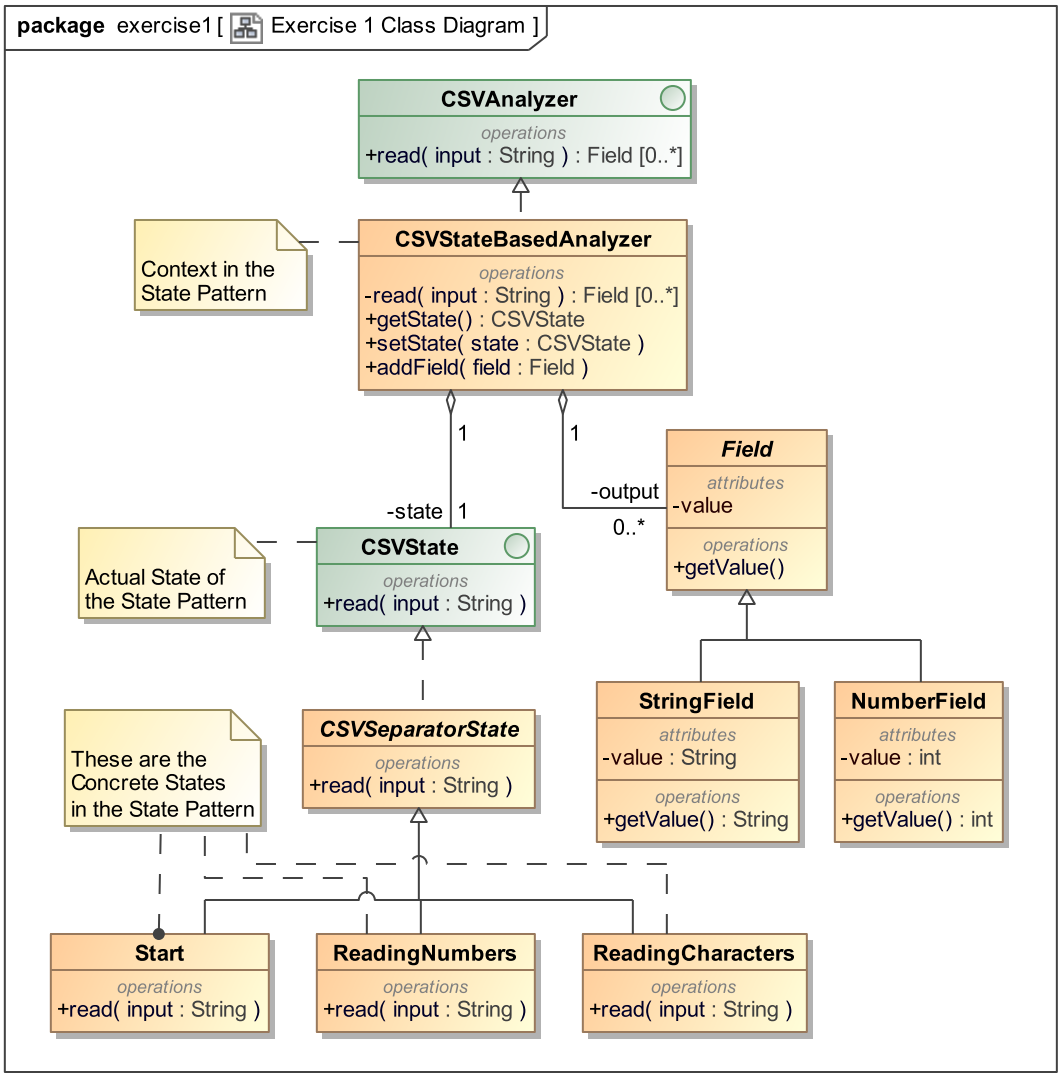
1 Introduction

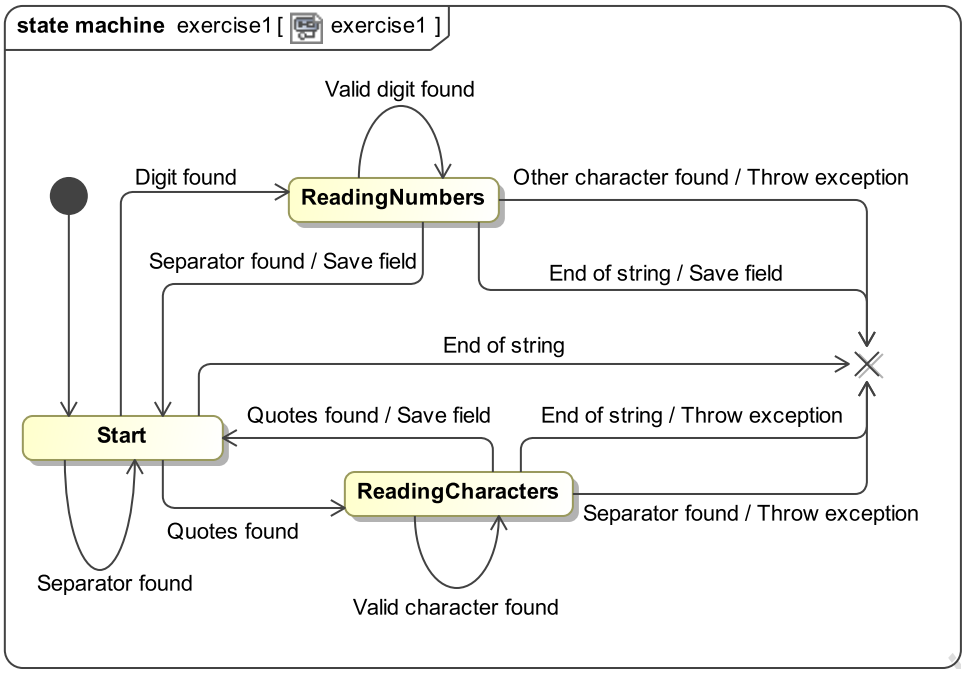
In order to perform an actual implementation of the proposed exercises, several compromises, i.e. design decisions, had to be made. In those cases where simplicity and extensibility were at conflict, we have chosen the latter.

Essentially, we have adhered as closely as possible to SOLID design principles, the details of which will be explained in each exercise's subsections.

2 CSV Data Analyzer

The CSV Data Analyzer we have built takes a String as input and returns a list of Field objects. The possible field types are strings or numbers.


package exercise1 [Exercise 1 Class Diagram]
CSVAnalyzer
operations
+read(input : String) : Field [0..*]
Context in the State Pattern
CSVStateBasedAnalyzer
operations
-read(input : String) : Field [0..*]
+getState() : CSVState
+setState(state : CSVState)
+addField(field : Field)
1
1
-state
1
-output
0..*
Field
attributes
-value
operations
+getValue()
Actual State of the State Pattern
CSVState
operations
+read(input : String)
StringField
attributes
-value : String
operations
+getValue() : String
NumberField
attributes
-value : int
operations
+getValue() : int
These are the Concrete States in the State Pattern
CSVSeparatorState
operations
+read(input : String)
Start
operations
+read(input : String)
ReadingNumbers
operations
+read(input : String)
ReadingCharacters
operations
+read(input : String)



2.1 Design principles

2.1.1 Single Responsibility Principle

The interfaces CSVState and Field grant that the code meets the SRP through increasing cohesion and lowering coupling.

Moreover, each object has a clearly defined responsibility, the ReadingNumbers and ReadingCharacters classes only read their type of fields.

An exception to this is the dependency of the reading states on their different types of fields. This means that an appropriate field must be created for each new kind of information read from a CSV. While this entails a small increase in coupling, we believe code simplicity is benefited to such an extent that uncoupling those classes would be unproductive. A field parser would then be needed, which requires detecting the type of parameters returned by the States, an overall bad practice.

Also for simplicity's sake, such dependencies are not shown on the UML Class Diagram.

2.1.2 Open-Closed Principle

All subclasses of CSVState and Field inherit their methods and override them when needed.

An example of this is the read() method in the CSVState interface, which needs to be different for the several kinds of functionality required to deal with a CSV-formatted String, while the overall form of a state remains the same.

2.1.3 Liskov Substitution Principle

The Liskov Substitution Principle is met since proper overriding is performed when needed and both the preconditions and post-conditions of the original functions are respected.

This can be clearly seen in the sub-classes of Field and CSVState, as the Analyzer can receive any type of Field or State, including future ones, because our concrete classes do not break the interface's contract.

2.1.4 Dependency Inversion Principle

Closely related to low coupling and SRP, the DIP is met due to the aforementioned interfaces on which the CSVAnalyzer relies.

2.1.5 Interface Segregation Principle

The simplicity of the task at hand renders the ISP of little importance.

2.1.6 Non-SOLID Principles

In order to meet the DRY (Don't Repeat Yourself) Principle, we developed a purpose-built abstract class called CSVSeparatorState. Though some would see it as over-engineering, we have decided to include it as a show of good practices with regard to extensibility, even if in this simple of an example it is not needed.

2.2 Design patterns

We have utilized the State Pattern with the Singleton Pattern for the implementation of the analyzer States, as can be seen in the Class Diagram.

The Singleton pattern can be thought of as part of the State Pattern, as it avoids creating several state instances.

The functioning of the state machine is described in the state diagram. The Start State acts as a selector, which can recognize the number and string fields, to then pass that information to the ReadingNumbers and ReadingCharacters states.

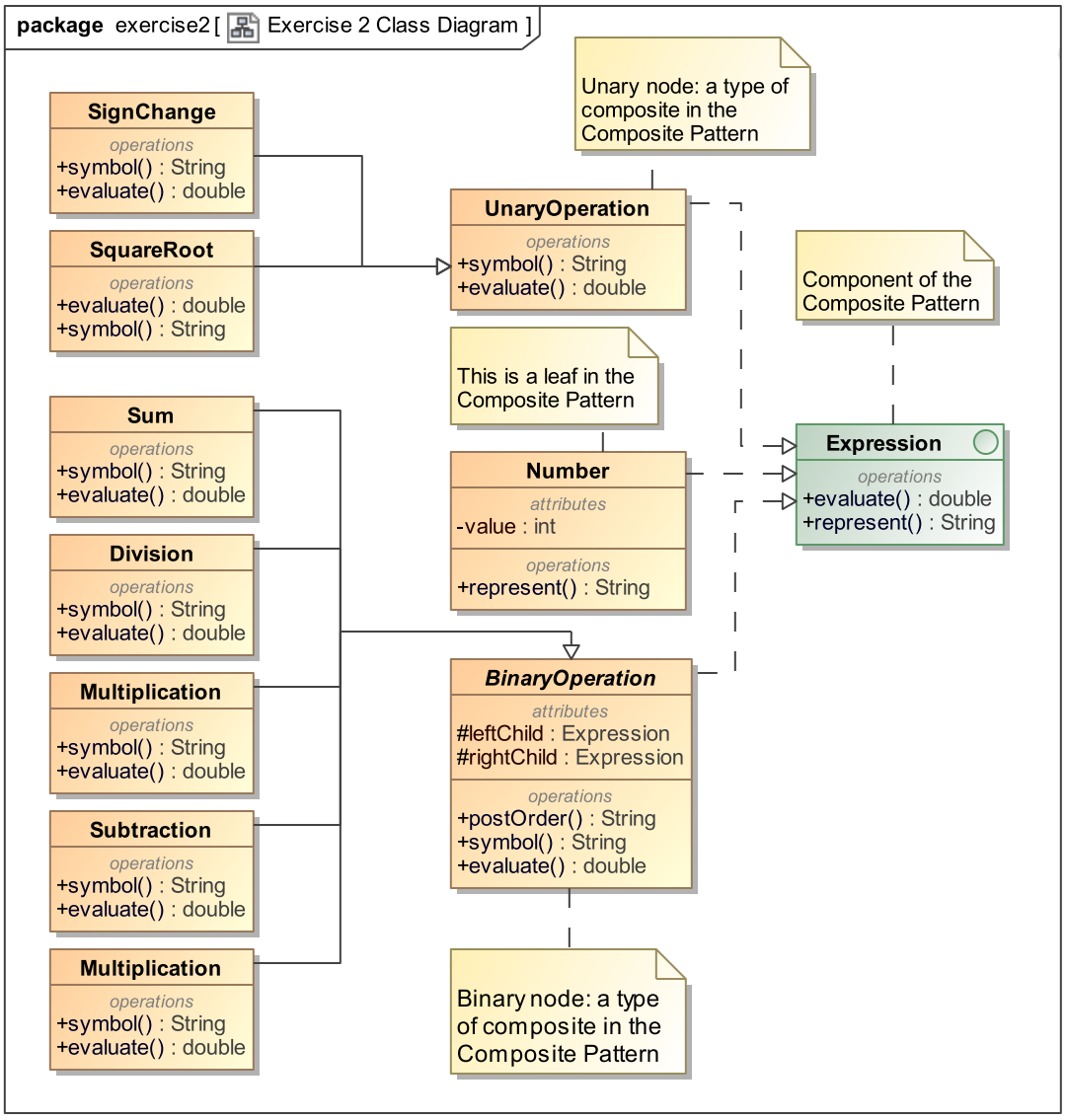
Those two latter states have the responsibility to add the read field to the list of fields on the analyzer, or, in the absence of a valid field, to throw an appropriate error. Our testing shows that each state handles its error handling individually and efficiently.

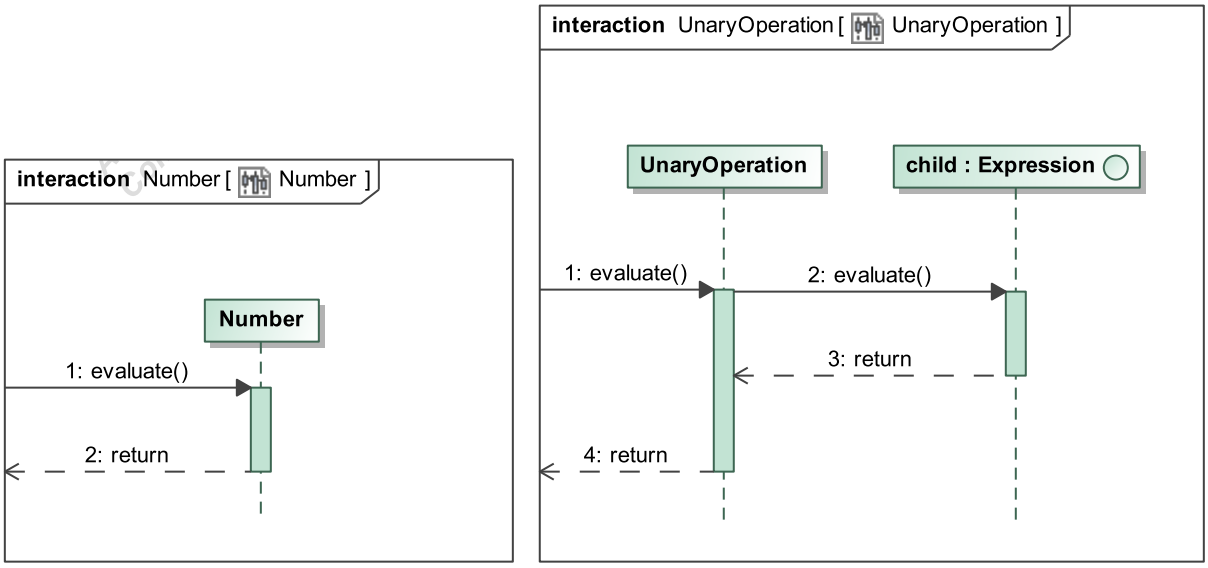
There is a certain similarity to the Strategy Pattern, as we left an Interface for future extension. On the contrary, the structure of the classes does not exactly match the pattern.

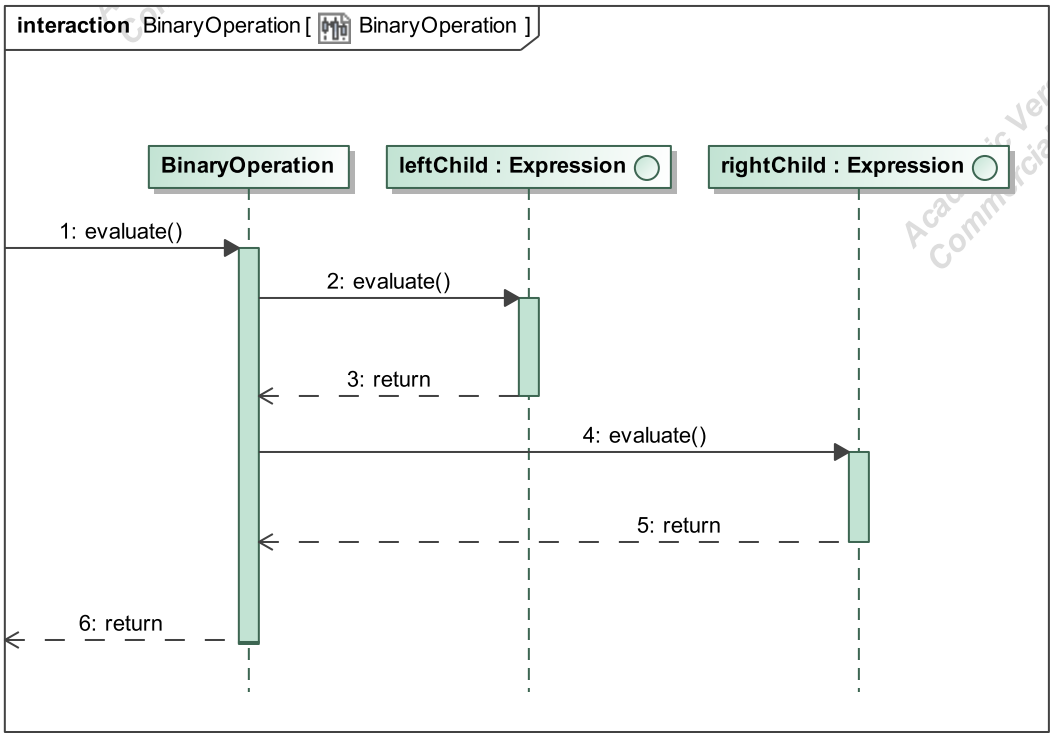
3 Arithmetic Analysis and Representation

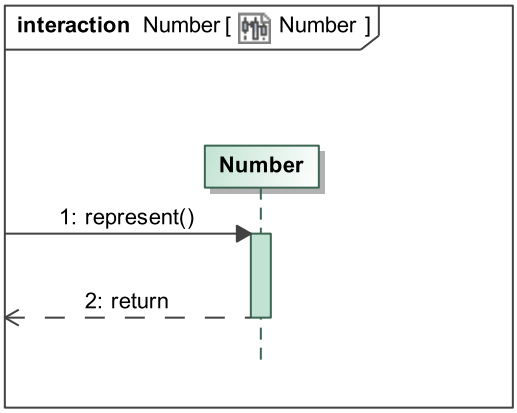
An Expression is a tree-like structured data type in the form of a class. It is highly encapsulated, to the point that no client is needed to evaluate nor represent the Arithmetic Expression.

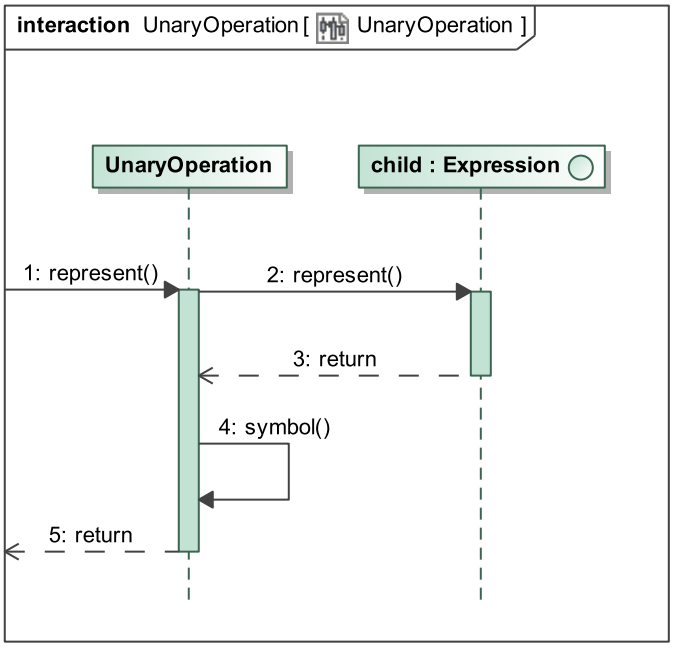
It is also extensible, as we have used a post-order tree traversal to generate the required Polish Notation. It would be trivial to also implement in-order and pre-order traversals of the tree.

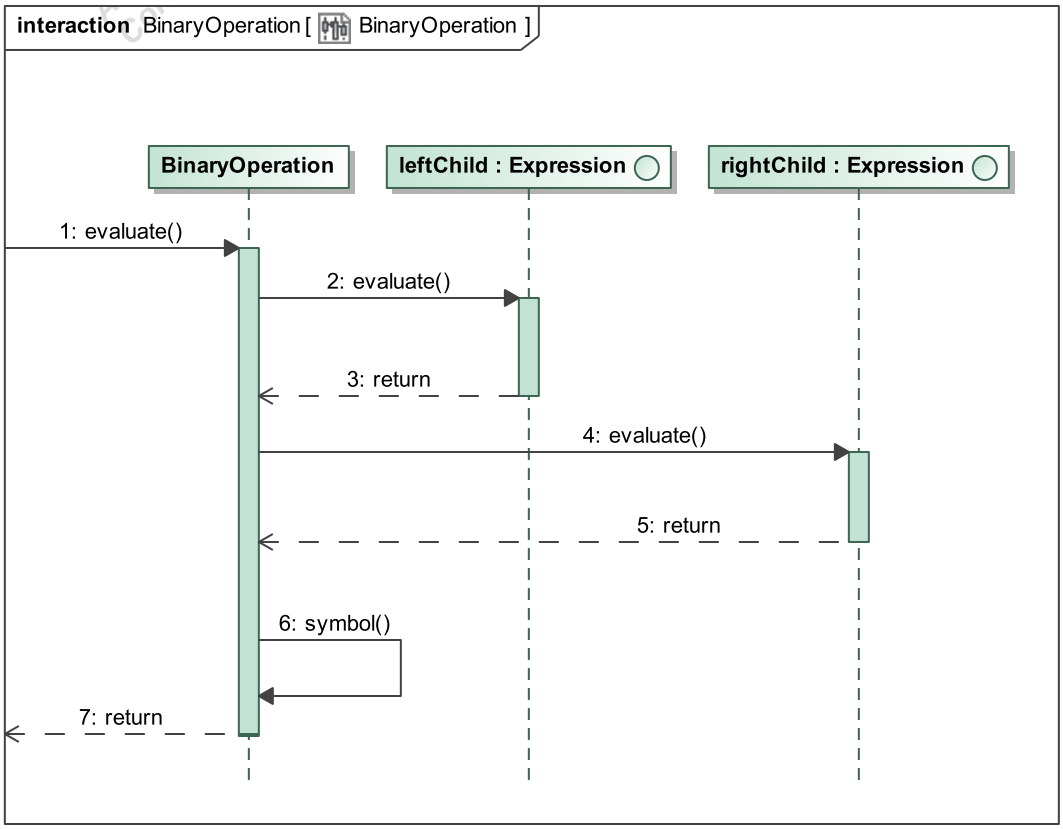

package exercise2 [Exercise 2 Class Diagram]
Unary node: a type of composite in the Composite Pattern
SignChange
operations
+symbol() : String
+evaluate() : double
SquareRoot
operations
+evaluate() : double
+symbol() : String
UnaryOperation
operations
+symbol() : String
+evaluate() : double
Component of the Composite Pattern
This is a leaf in the Composite Pattern
Sum
operations
+symbol() : String
+evaluate() : double
Number
attributes
-value : int
operations
+represent() : String
Expression
operations
+evaluate() : double
+represent() : String
Division
operations
+symbol() : String
+evaluate() : double
Multiplication
operations
+symbol() : String
+evaluate() : double
BinaryOperation
attributes
#leftChild : Expression
#rightChild : Expression
operations
+postOrder() : String
+symbol() : String
+evaluate() : double
Subtraction
operations
+symbol() : String
+evaluate() : double
Multiplication
operations
+symbol() : String
+evaluate() : double
Binary node: a type of composite in the Composite Pattern


interaction Number [Number]
Number
1: evaluate()
2: return
interaction UnaryOperation [UnaryOperation]
UnaryOperation
child : Expression
1: evaluate()
2: evaluate()
3: return
4: return


interaction BinaryOperation [BinaryOperation]
BinaryOperation
leftChild : Expression
rightChild : Expression
1: evaluate()
2: evaluate()
3: return
4: evaluate()
5: return
6: return


interaction Number [Number]
Number
1: represent()
2: return


interaction UnaryOperation [UnaryOperation]
UnaryOperation
child : Expression
1: represent()
2: represent()
3: return
4: symbol()
5: return



3.1 Design Principles

3.1.1 Single Responsibility Principle

Each class serves a clearly defined purpose: the BinaryOperation, UnaryOperation and Expression interfaces increase cohesion and achieve low coupling. Also, there is a class for each operation and a single operation per class.

All of this means the SRP is met.

3.1.2 Open-Closed Principle

All subclasses of BinaryOperation and UnaryOperation inherit and override, only when needed, their methods. An example is the symbol() method, which greatly simplifies the representing mechanism and allows for great extensibility.

3.1.3 Liskov Substitution Principle

The Liskov Substitution Principle is met with great accuracy thanks to the recursive nature of our structure. The children of nodes can be any kind of expression, as a result of the compliance to the expectations set by the interface. That means nodes can depend on children's ability to correctly resolve an evaluate() or represent() petition. This exemplifies the Design by Contract (DBC) proposed by Bertrand Meyer.

3.1.4 Dependency Inversion Principle

As already mentioned, nodes depend on interfaces rather than concrete classes.

3.1.5 Interface Segregation Principle

Once again, the simplicity of the exercise does not call for interface segregation, as the amount of different functionalities is rather limited.

3.1.6 Non-SOLID Principles

Developing a client class that operated on Arithmetic Expressions was our first approach. Despite this, and upon consideration, we realized that the architecture of this design not only is not prone to, but rather completely lacks the need for it. As a prime example of one of Object Orientation's paramount features, expressions are self-contained to the point that they can perform all functionalities required on their own.

3.2 Design Patterns

The Composite Pattern is met in our implementation to grant the user an efficient, self-contained and extensible way of operating with arithmetic expressions.

The recursive nature of the tree data structure makes extensibility a prime feature of this Pattern. We have also chosen to make the Expression class immutable for safety in order to comply with the Immutable Pattern.