



Project Report

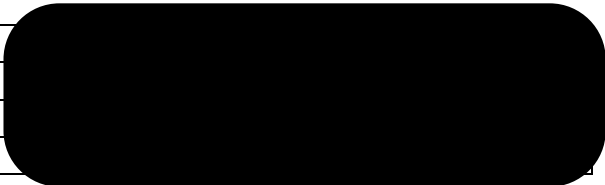
Name	
Jun Lin Chen	
Vishal Malik	
Ragheb Abunahla	

Table of Contents

LE EECS 3311 Fall 2017 Project Report	1
Table of Contents	2
Design Objective	3
BON Diagram.....	4
All Relevant Classes.....	4
Expression Language Structure	5
Three Language Operations.....	6
Program Structure	7
Main Structure.....	7
Expression Folder	7
Operation Folder.....	9
Utility Folder	9
Design Principles	11
Information Hiding.....	11
Single Choice Principle	11
Open/Close Principle.....	12
Uniform Access Principle.....	13
Design Patterns and Ideas	13
Separation of Concerns	13
Void Safety	13
Visitor Pattern.....	13
Appendixes	14
In Class Features Relations	14

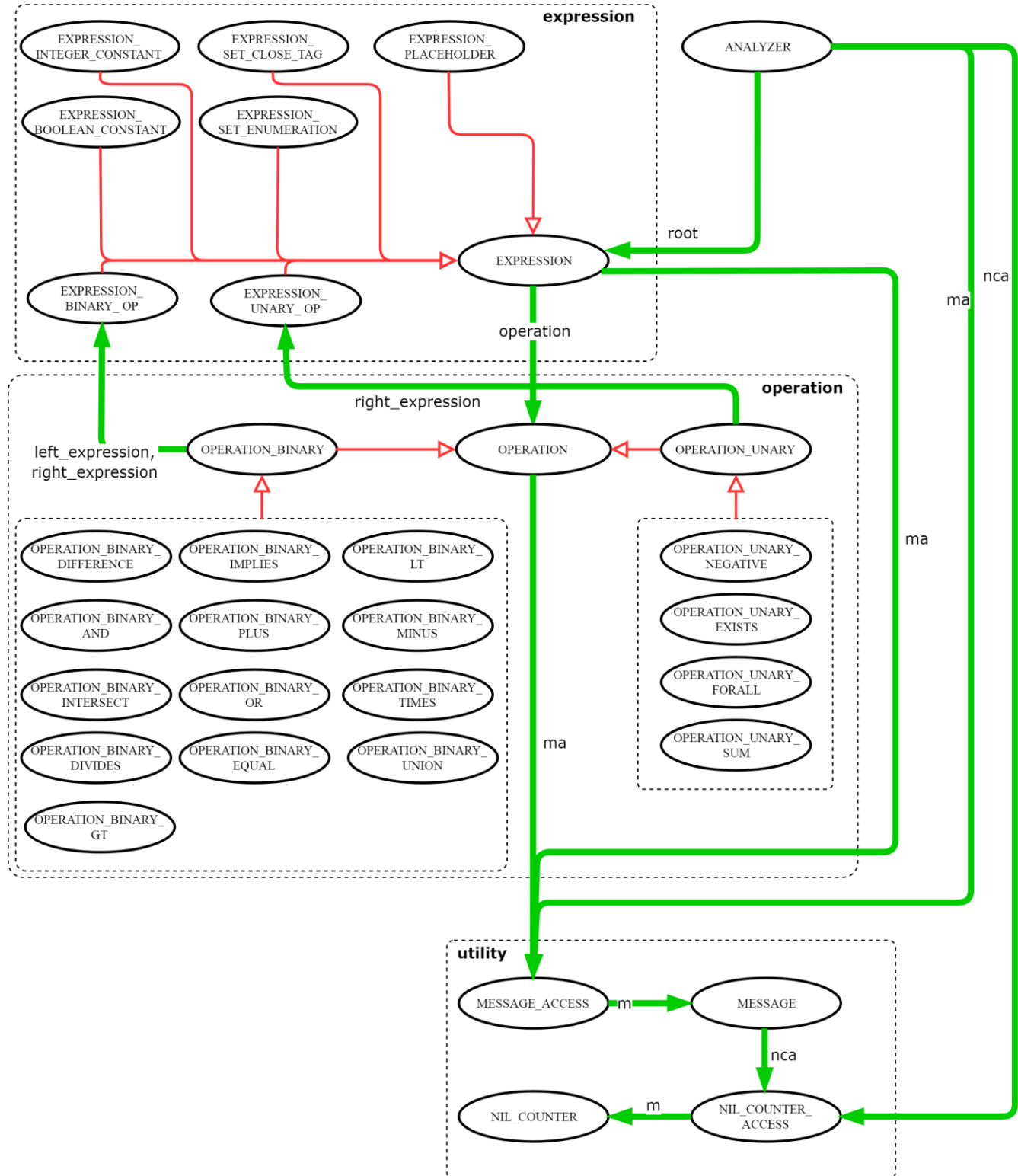
Design Objective

Our design objective is to build a stable analyzer for a small expression language that with high expansibility and high readability under the Eiffel Testing Framework.

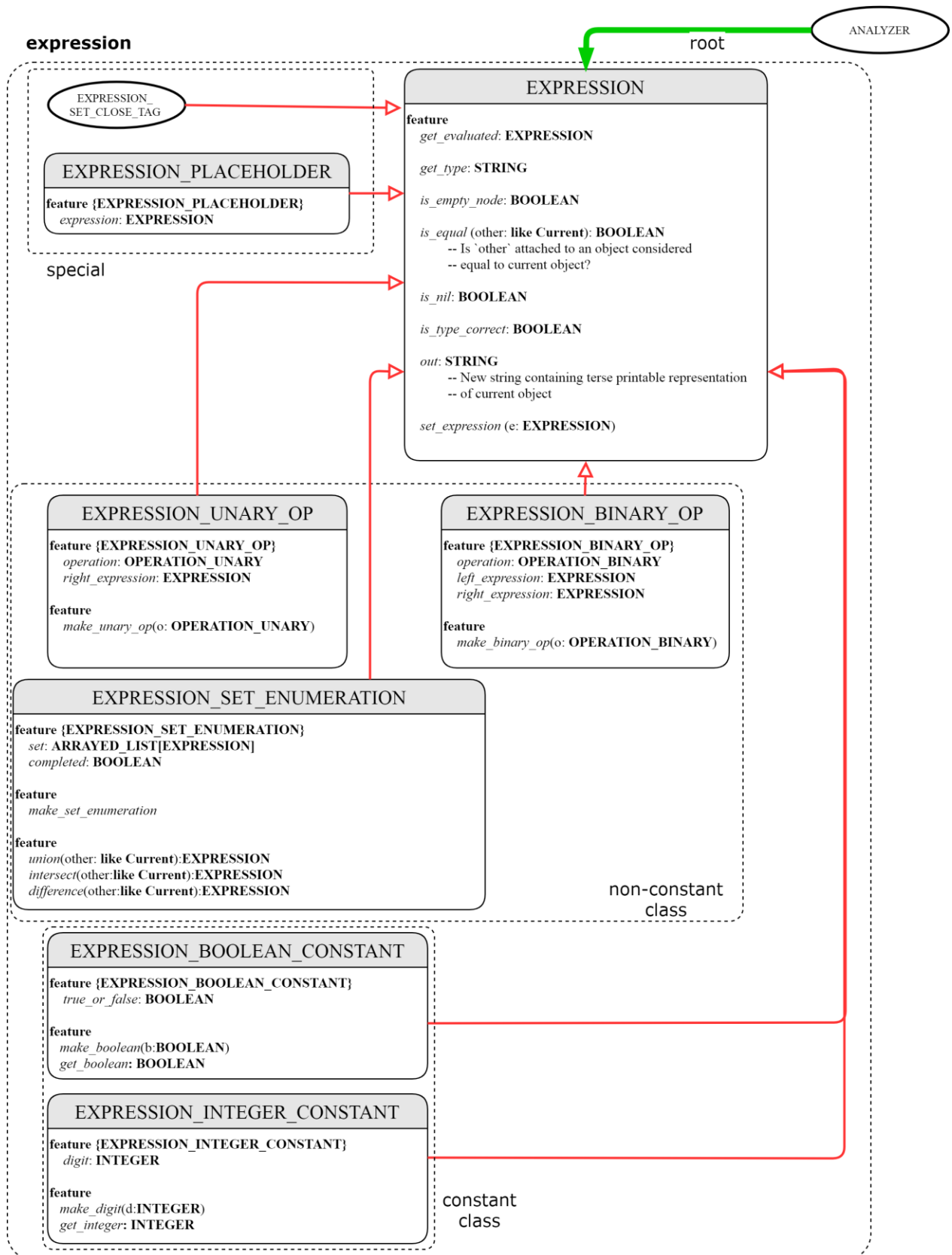
The design principle is the methodology tells us how to approach the design objective. If one design principle is controversial to another design principle, we will use the design objective to judge which principle to be implemented.

BON Diagram

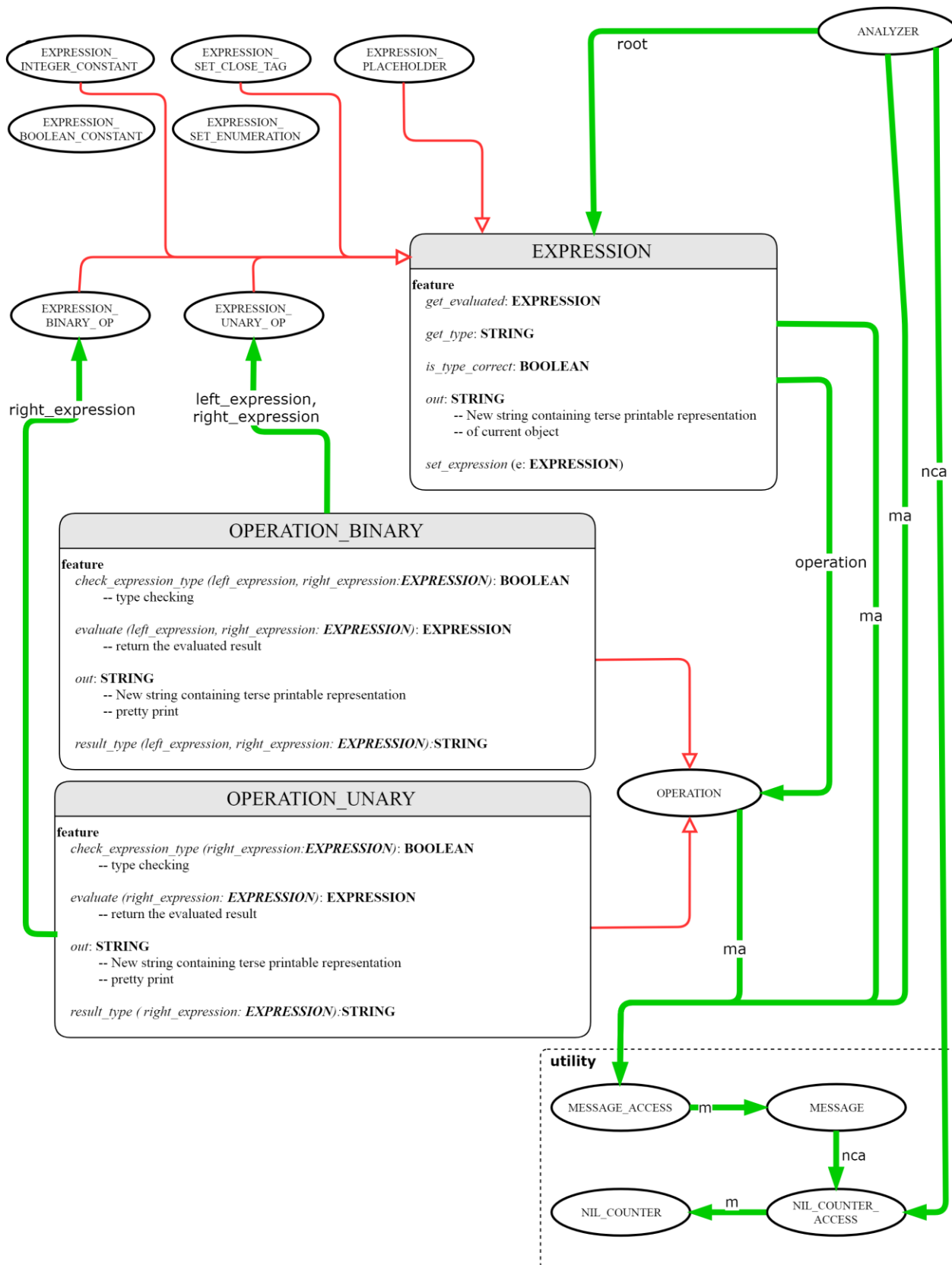
All Relevant Classes



Expression Language Structure



Three Language Operations



Program Structure

Main Structure

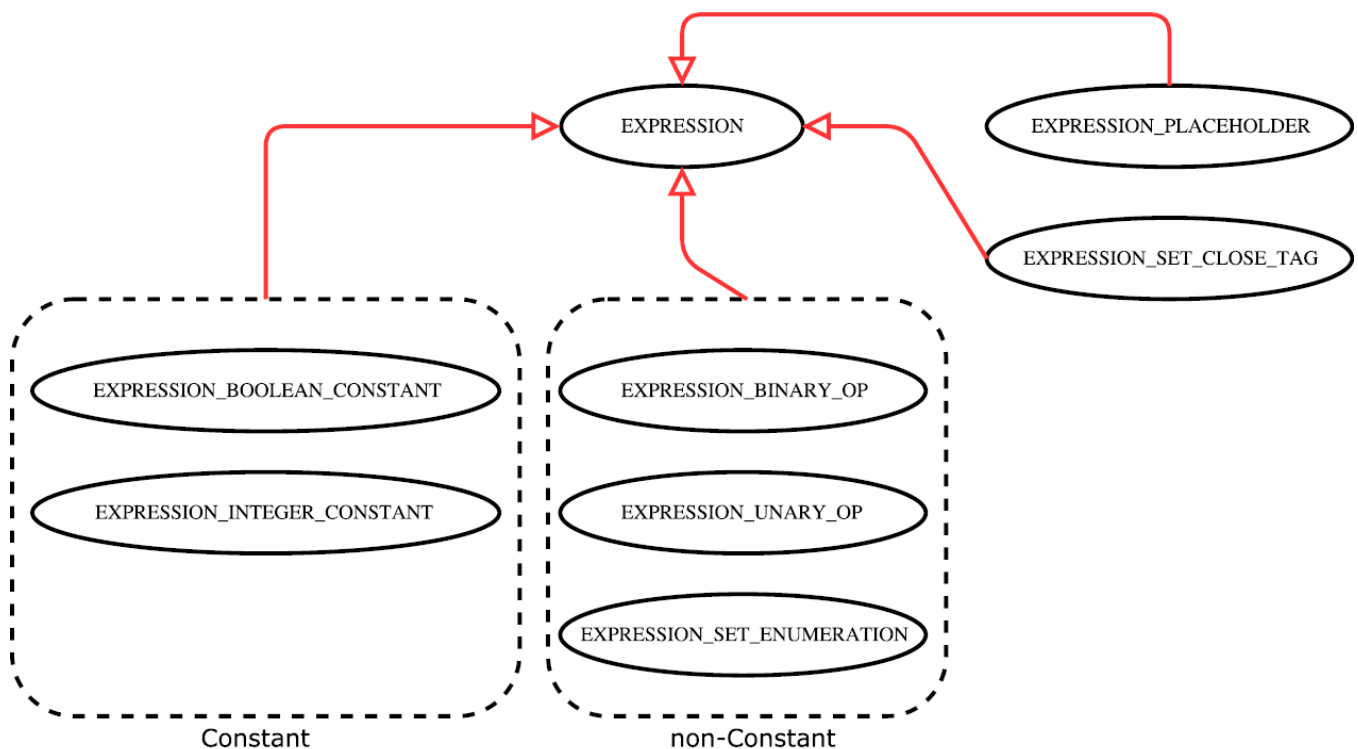
Command handling is given by the ETF framework. We do not need to care about it too much. In our design, we did the minimal changes on the ETF framework. The classes in the `user_commands` folder merely invoke a single operation on the model. After each execution, the return value will be provided to the output of the model. Then the ETF framework prints the output of the model by default.

We implement this analyzer using three group of classes.

- **Expression** handles the grammar of the expression language.
- **Operation** handles all the terminals (except brackets) in the expression language.
- **Utility** contains some classes that support three operations on the expression language.

Each group is explained in detail below.

Expression Folder



The `EXPRESSION` class is the parent class of all the classes in this folder. Because it has not been specified, it is also being used as a representation of an empty expression (`VOID`).

The `EXPRESSION_SET_CLOSE_TAG` class is one special class in this folder. It identifies the ending of a set. By using this class, we can treat the “`close_set_enumeration`” command a regular command that can be handled recursively.

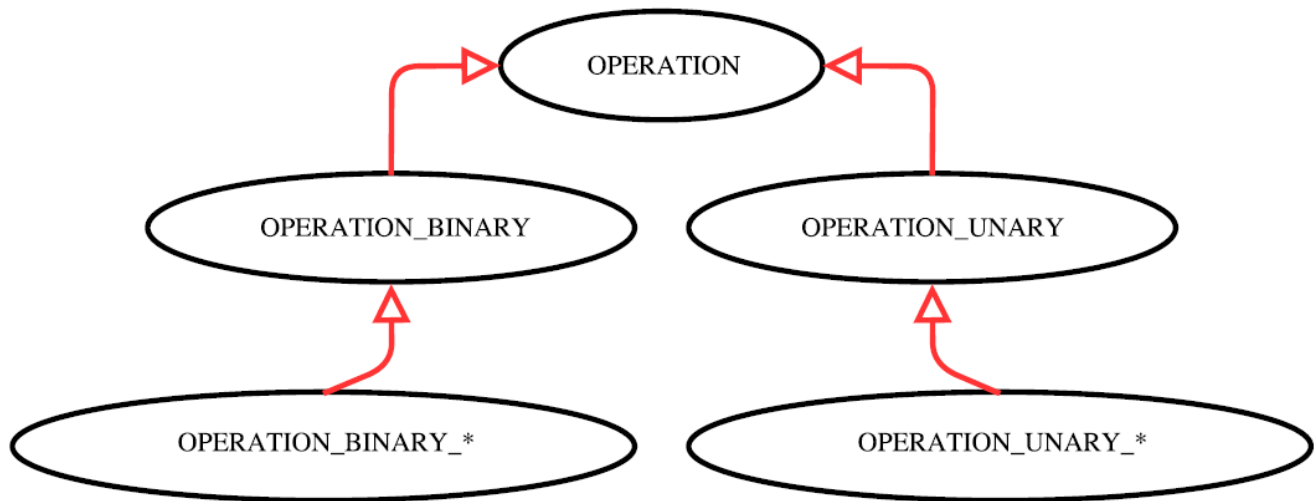
The `EXPRESSION_PLACEHOLDER` class, like a glue connecting different component, are the edges of the parsing tree. By default, the substructure hold by this class is initialized to `EXPRESSION` object, which means it is empty.

Other than these classes. There are different types of classes:

1. Classes for constants
 - `EXPRESSION_INTEGER_CONSTANT`
 - `EXPRESSION_BOOLEAN_CONSTANT`
2. Classes for non-constant
 - `EXPRESSION_BINARY_OP`
 - `EXPRESSION_UNARY_OP`
 - `EXPRESSION_SET_ENUMERATION`

All the non-constant classes have substructures (`EXPRESSION_PLACEHOLDER`) that allow expending following their rules.

Operation Folder



This group of classes define all the functional terminal symbols in this language. There are two types of operations; one is unary operation which accepts one expression on one side, one is binary operation which accepts two expressions on both sides.

It defines operation the followings:

1. Pretty Print
2. Type Checking
3. Math Calculation
4. Possible data type of the result of the calculation

Utility Folder

Because operations are handled recursively, considering that we have the following requirements need to be accomplished:

1. We need to print “?” instead of “nil” when it is the first appearance of “nil”.
2. Some error messages are produced within the recursive procedures (e.g. type error, division by zero)
3. We need to know whether the user command is taken effect.

Above all, we implement two singleton classes to help us to accomplish these requirements.

- **NIL_COUNTER**

It was initialized to true. During the recursive procedure, if it is touched or queried, it will be set to false. It is a crucial feature for pretty print and the set expression operation. If it is still true after the execution, that means the operation has not yet be carried out. Hence, something might go wrong. The MESSAGE class will take care of this situation.

- **MESSAGE**

This class takes control of all the error message. By implementing it in a singleton, it can be accessed at any point during the recursive handling procedures. It updates the status of the model after each execution.

Design Principles

Information Hiding

For the model, we limited the access from outside to only:

1. The status of the model
2. Supported user features

For the expression classes, its implementation is hidden so that only the expression itself can check. By using separation of concerns, each group of classes (as [described above in the program structure section](#)) are designed to be isolated. Hence, they satisfied information hiding principle.

In this project, the implementation of the model is hidden and may be changed. The application interface of the model, as we implemented, is public but stable.

Single Choice Principle

The Single Choice Principle can be seen following in regions of the code where we established inheritance. If you refer to [the BON diagram above](#), you'll be able to witness the relationship between `EXPRESSION` and its children classes as well as `OPERATION` and its children classes. There is no duplicate code fragment in this project.

This inheritance relationship allows us to make particular features for each class and we isolate the responsibilities accordingly. For example, if something goes wrong with a feature in the parent class `EXPRESSION`, we only have to modify that feature of `EXPRESSION` because all its subclasses will just inherit the modified fixed feature. This takes away from tedious time and effort spent going through all the code and making modifications. It also allows for a cleaner structure since tasks are appointed responsibly.

And also, whenever we want to add a new operation or a new expression syntax (production) we can just make a new subclass, and this is us obeying the single choice principle. Since our structure does not change but only the operations do we are allowed to obey the Single Choice Principle with the visitor pattern. Although Eiffel supports multiple inheritances, we are not using it in our project.

Open/Close Principle

This program satisfies this principle. This program is open for extension but closed for modification. All the module that is available for use by other module are well defined. In order to add a new feature, we just need to add new classes rather than modifying existing classes.

To implement a new operation, you just need to create a new OPERATION subclass based on what the operation is meant to do and redefine the following features:

- evaluate
- result_type
- check_expression_type
- out

To implement a new expression syntax (production), you just need to create a new EXPRESSION subclass based on what the expression is meant to represent and redefine the following features:

- is_equal
- is_type_correct
- set_expression
- get_type
- get_evaluated
- out

To implement a new data type, you just need to create a constant class with a customized implementation of that data type inside the class which is similar to the `EXPRESSION_BOOLEAN_CONSTANT` and the `EXPRESSION_INTEGER_CONSTANT`.

Above all, to add a new feature, we do not need to modify existing code, we just need to add new classes. By following these rules, we can add all the operations and expressions required by the project instruction at a considerable speed.

Uniform Access Principle

This principle is enforced by the Eiffel language. Clients should not have to know that a feature is implemented by routines or attributes. The client does not care what is happening to the code on the back end.

In this project, for example, the client just knows that the feature `get_evaluation` can return the simplified version of the expression, but they do not need to know how complicated it was implemented behind the scenes. Behind the scenes, the implementation is complicated. For constant expression classes, they just return itself. But for non-constant expression classes, it involves a series of recursive function calls to generate the simplified expression.

Design Patterns and Ideas

Separation of Concerns

That is the reason why we separate this program into three parts as described in [the program structure section](#) above. Furthermore, we do not care about how the ETF framework handles the user command. We only care about how to implement the model.

Void Safety

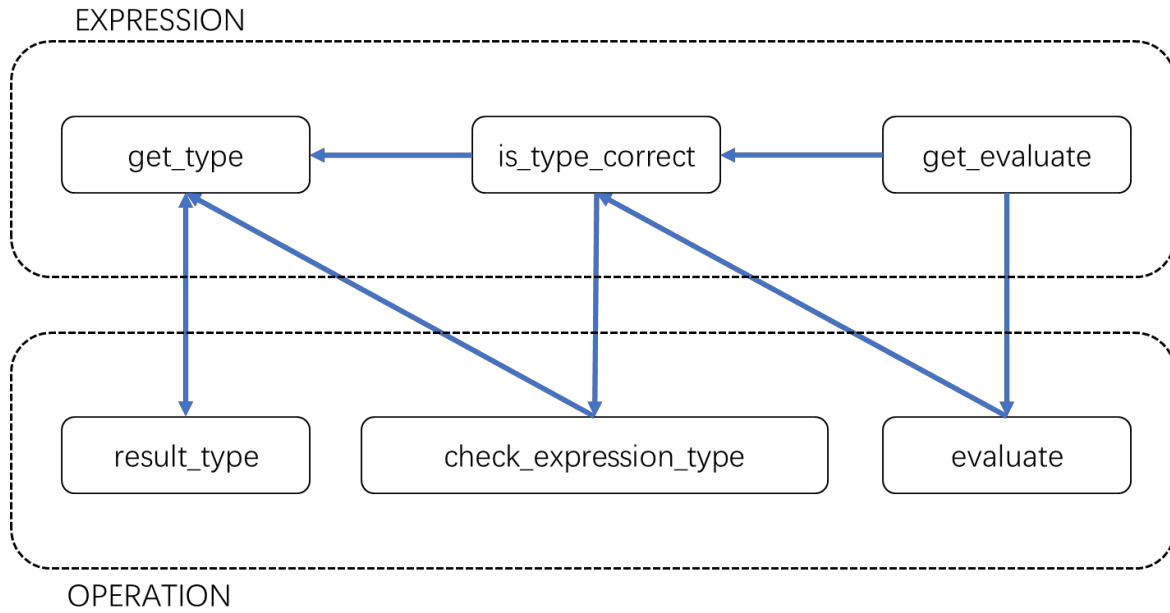
In our code, we avoid using any detachable returns or variables such that we never allow any void variable to exist at runtime. Although we never use void, we regard the raw `EXPRESSION` object (not its subclasses' object) as an empty object.

Visitor Pattern

This program uses visitor pattern for type checking operation and evaluation. The `OPERATION` classes are the visitors that visit all the nodes in the parsing tree to check the type of each node or evaluate the result.

Appendixes

In Class Features Relations



In order to be evaluated, the expression must be type correct. So, there is an internal two-ways client-supplier relation between **EXPRESSION** and **OPERATION** class.