Applying the Software Quality principles by application development on the subject of Big Numbers Operations

Apachiței Maria-Luisa

Master of Computational Optimization, first year

Bumbu Ana-Maria

Master of Software Engineering, first year

Munteanu Rareș-Costin

Master of Computational Optimization, first year

Vîrvarei Alexandru

Master of Computational Optimization, first year

23 Mai 2021

The proposed project within the discipline *Software Quality* is developed for allowing the operations between very big numbers - positive integers. The requirements regarding the functionality are limited to the basic operations: addition, subtraction, multiplication, division, power and square root. After the first phase of application development, the team members focused on testing. Firstly, we've performed *unit testing*, after that making use of *assertions*, which implies the direct access to the code. All these mechanisms are performed in order to draw the conclusions regarding the software quality.

Keywords: Big Numbers Operations, Unit Testing, Assertions, Coverage, Software Quality

1. Introduction

The aim of this paper is to make a documentation of the proposed application - designed to manage the operations between Big Numbers. We will present its architecture, for specifying the core guidelines. The goal was to create a working version of the program, not necessarily fully stable or error-free, on which testing techniques will subsequently be applied.

Then, we've explored the mechanisms used for evaluating the software quality and we've concluded on using the unit testing and the assertions, during different phases of our project.

The results returned by the testing tools are analyzed and that's how we will discover some future paths that will help us to improve the proposed software.

1.1 Modelling of the Problem

In order to use the testing principles, we had to develop an application which allows carrying out operations with very big numbers - positive integers.

The architecture impose some guidelines and restrictions. As discussed above, the problem is reduced to the following basic operations: addition, subtraction, multiplication, division (integer), power and square root.

There was needed an interface in other to interact with users. The application needs to perform the following flow:

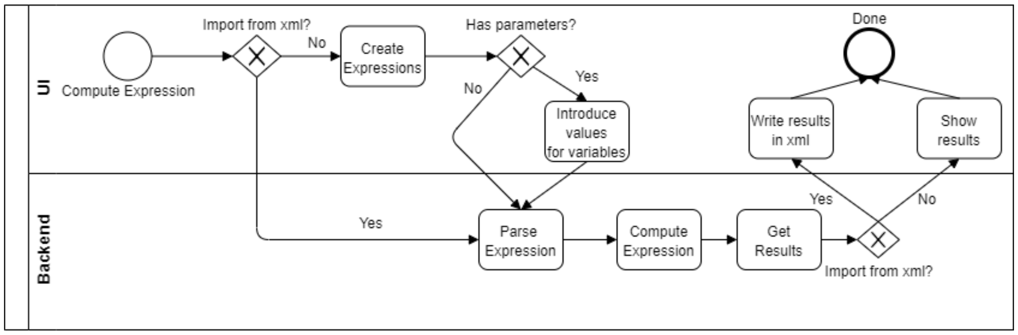


Figure 1. BPMN diagram - the working flow

- The user can request the computation of an arithmetic expression (e.g. *(a + b) * a*) and it can provide additional variables, with the desired initialization.
- The program displays the results of each step in the computation. Any incorrect outcome (negative result of a subtraction, division by 0) must be signaled explicitly, which results in immediately terminating the computation.
- The expression and the values of the variables are entered through a user dialog, which allows creating the expression as a composition of basic operations. The results are also displayed in the same dialog window.

- The data input is read from an XML file. Also, the results are written into an XML file.

The implementation was development using the programming language Python.

So, the numbers are stored as lists - aka. arrays, the programming language already giving us a way of controlling the number size. As required, we don't make use of library functions for computing the results of operations, for parsing the expressions or for processing the errors. The code is provided by programmers - with a little help from the basic modules.

For completing the task, a dedicated section will describe the testing mechanisms used for measuring the software quality.

2. Architecture

The core units of the application will be described in order to be able to write the proper unit cases used in the testing phases.

- **Operations** class contains the core implementations of the arithmetic operations: addition, difference, multiplication, divide, power and square root. There are also helping functions used inside core operations.
- **BigNumber** class overrides the basic arithmetic operators with functions defined in Operations class.
- **Node** class holds the representation of a node inside an syntax tree.
- **Parser** class is used to create the syntax tree and also switch parameters with their correspondence values.
- **Computation** class used all the above units for parsing and compute expression result.
- **MainWindow** and **ResultWindows** classes are used for displaying the results on the interface.

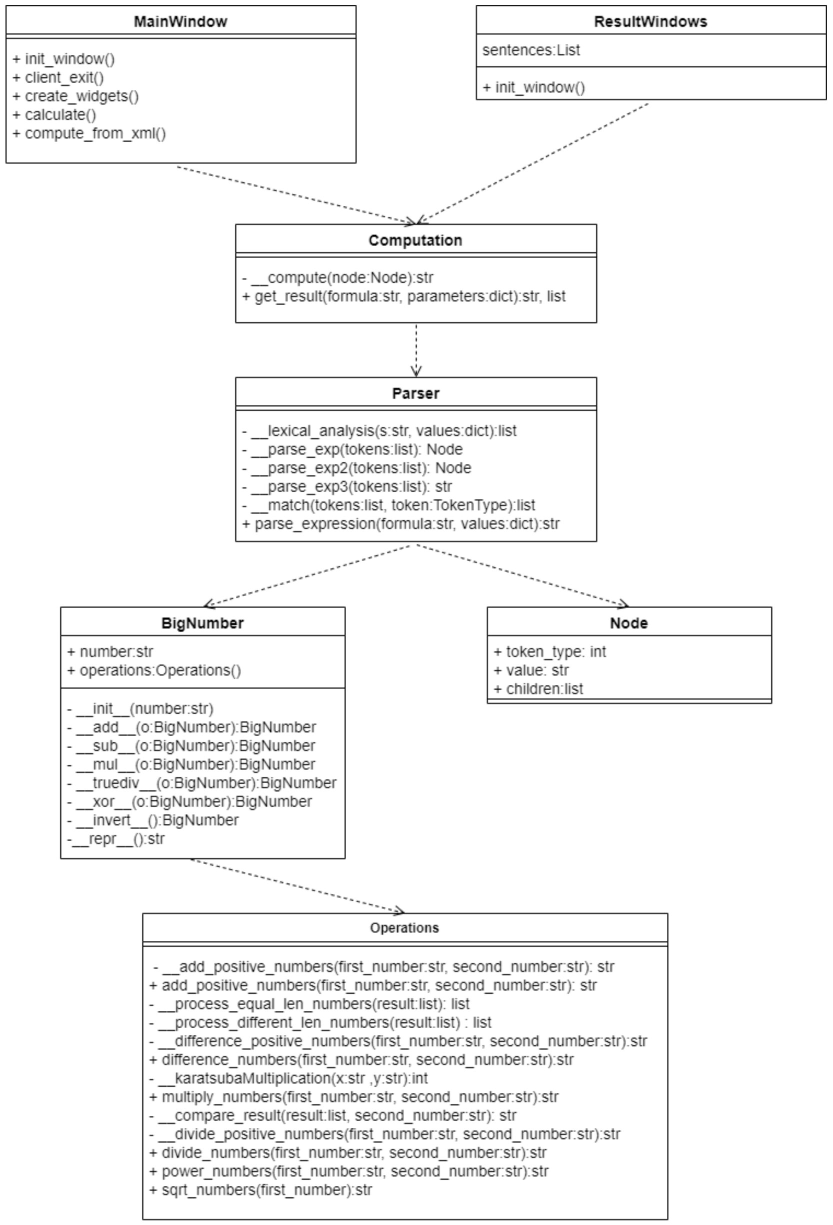

MainWindow
+ init_window()
+ client_exit()
+ create_widgets()
+ calculate()
+ compute_from_xml()
ResultWindows
sentences:List
+ init_window()
Computation
- __compute(node:Node):str
+ get_result(formula:str, parameters:dict):str, list
Parser
- __lexical_analysis(s:str, values:dict):list
- __parse_exp(tokens:list): Node
- __parse_exp2(tokens:list): Node
- __parse_exp3(tokens:list): str
- __match(tokens:list, token:TokenType):list
+ parse_expression(formula:str, values:dict):str
BigNumber
+ number:str
+ operations:Operations()
- __init__(number:str)
- __add__(o:BigNumber):BigNumber
- __sub__(o:BigNumber):BigNumber
- __mul__(o:BigNumber):BigNumber
- __truediv__(o:BigNumber):BigNumber
- __xor__(o:BigNumber):BigNumber
- __invert__():BigNumber
-__repr__():str
Node
+ token_type: int
+ value: str
+ children:list
Operations
- __add_positive_numbers(first_number:str, second_number:str): str
+ add_positive_numbers(first_number:str, second_number:str): str
- __process_equal_len_numbers(result:list): list
- __process_different_len_numbers(result:list) : list
- __difference_positive_numbers(first_number:str, second_number:str):str
+ difference_numbers(first_number:str, second_number:str):str
- __karatsubaMultiplication(x:str ,y:str):int
+ multiply_numbers(first_number:str, second_number:str):str
- __compare_result(result:list, second_number:str): str
- __divide_positive_numbers(first_number:str, second_number:str):str
+ divide_numbers(first_number:str, second_number:str):str
+ power_numbers(first_number:str, second_number:str):str
+ sqrt_numbers(first_number):str

Figure 2. UML diagram - the core units of the application

3. Contribution per Team Member

The architecture of our project was previously designed, before the application development, such that the core units were equally distributed between the team members. During the next three phases, each team member was responsable of its classes (aka. units) by applying the requirements specific on these: the unit testing, the appliance of assertions, simultaneously writing the documentation.

The manager of each core unit:

1. *ResultWindows and MainWindow classes; Coverage Module* - Vîrvarei Alexandru
2. *BigNumber and Operations classes* - Apachiței Maria-Luisa, Munteanu Rareș-Costin
3. *Parser and Computation classes* - Bumbu Ana-Maria

4. Testing Mechanisms

The “unittest” unit testing framework has been built into the Python standard library since version 2.1 and is similar with the majority of unit testing frameworks in other languages. This framework has some important requirements for writing and executing tests:

- Tests need to be into classes as methods.
- Usage of a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement.

The basic building blocks of unit testing are test cases, scenarios that must be set up and checked for correctness. In unittest, test cases are represented by unittest.TestCase instances. Tests can be numerous, and their set-up can be repetitive. For this reason, can be implemented a method called setUp(), which the testing framework will automatically call for every single test run.

The “unittest.mock” is a library for mocking in Python. As of Python 3.3, it is available in the standard library. Allows to replace parts of the system under test with mock objects and make assertions about how they have been used. Mock objects create all attributes and methods as they are accessed and store details of how they have been used. Accessing the same attribute will always return the same mock. Mocks allow programmers to make assertions about what their code has done to them. The patch() decorators make it easy to temporarily replace classes in a particular module with a Mock object.

The “coverage.py” is a tool for measuring code coverage of Python programs. It monitors programs, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not. Coverage measurement is typically used to evaluate the effectiveness of tests. It can show which parts of the code are being exercised by tests, and which are not.

The most common way to do defensive programming is to add assertions to code so that it checks itself as it runs. An assertion is simply a statement that something must be true at a certain point in a program. When Python sees one, it evaluates the assertion's condition. If it's true, Python does nothing, but if it's false, Python halts the program immediately and prints the error message if one is provided.

Assertions fall into three categories:

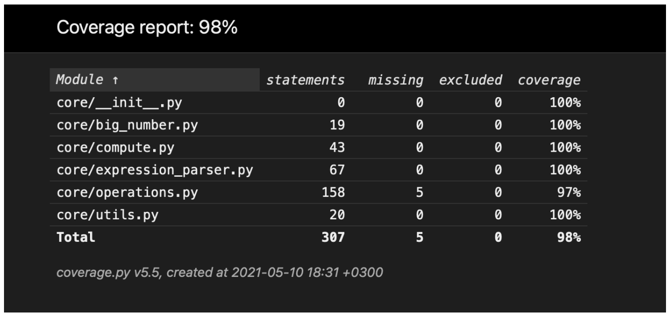
- A precondition is something that must be true at the start of a function in order for it to work correctly.
- A postcondition is something that the function guarantees is true when it finishes.
- An invariant is something that is always true at a particular point inside a piece of code.

Assertions aren't just about catching errors, they also help people understand programs. Each assertion gives the person reading the program a chance to check that their understanding matches what the code is doing.

5. Experiments, results and discussions

5.1 Unit Testing

We have run the unittesting module with the purpose of improving our tests quality in order the achieve a code coverage percent as high as possible. Doing so, we had a pretty strong insight of what we should insist on testing, and what we can improve in our code.


Coverage report: 98%

Module ↑	statements	missing	excluded	coverage
core/__init__.py	0	0	0	100%
core/big_number.py	19	0	0	100%
core/compute.py	43	0	0	100%
core/expression_parser.py	67	0	0	100%
core/operations.py	158	5	0	97%
core/utils.py	20	0	0	100%
Total	**307**	**5**	**0**	**98%**

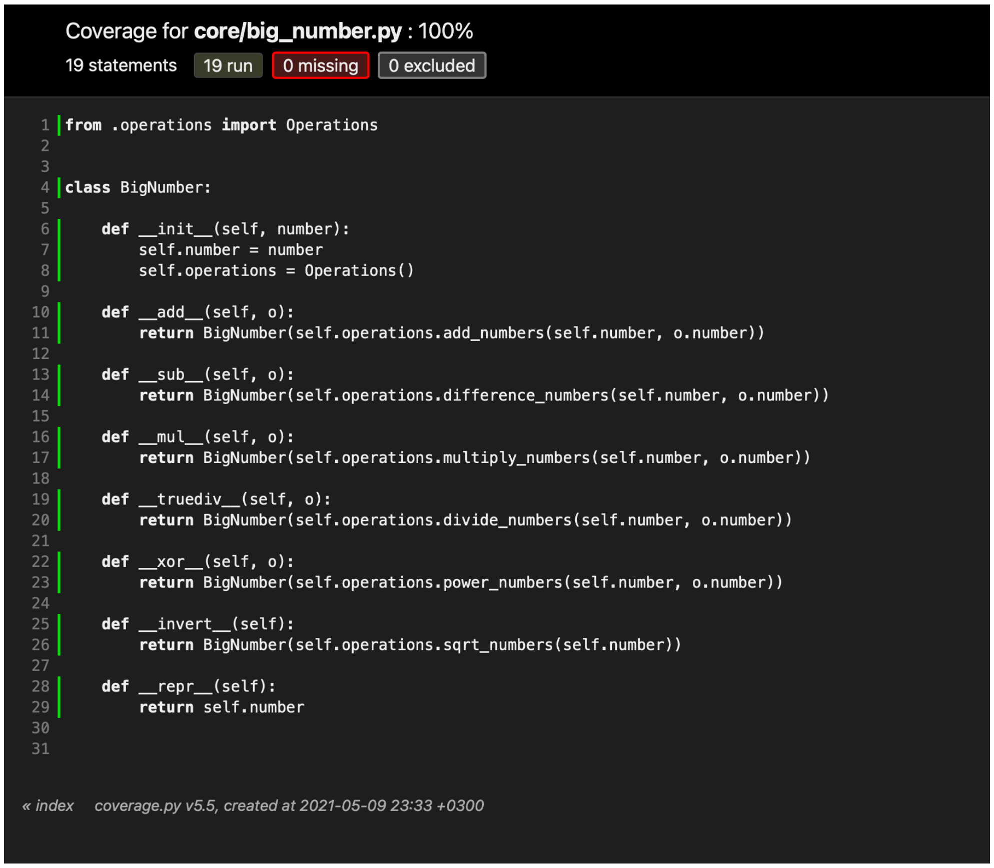
coverage.py v5.5, created at 2021-05-10 18:31 +0300

Figure 3. Code coverage report

5.1.1 Code Coverage

Figure 3 shows the coverage report we have obtained. We can see that each module has its own coverage score.

We notice that "operations.py" achieved only 97% coverage. This is because there is a code branch that can never execute, and we were able to discover it thanks to this approach. The coverage module generates extremely useful reports, which offer valuable information through an html interface.



Coverage for **core/big_number.py** : 100%

19 statements 19 run 0 missing 0 excluded

```
from .operations import Operations


class BigNumber:

    def __init__(self, number):
        self.number = number
        self.operations = Operations()

    def __add__(self, o):
        return BigNumber(self.operations.add_numbers(self.number, o.number))

    def __sub__(self, o):
        return BigNumber(self.operations.difference_numbers(self.number, o.number))

    def __mul__(self, o):
        return BigNumber(self.operations.multiply_numbers(self.number, o.number))

    def __truediv__(self, o):
        return BigNumber(self.operations.divide_numbers(self.number, o.number))

    def __xor__(self, o):
        return BigNumber(self.operations.power_numbers(self.number, o.number))

    def __invert__(self):
        return BigNumber(self.operations.sqrt_numbers(self.number))

    def __repr__(self):
        return self.number
```

« index coverage.py v5.5, created at 2021-05-09 23:33 +0300

Figure 4. Code coverage for big_number.py

Figure 4 is an example of coverage report for "big_number.py" module. A green bar at the beginning of the line means that the tests cover that line of code, while a red one means it is not covered.

5.1.2 The tested scenarios

BigNumber class

We've performed tests over the BigNumber class in order to check the computation results and the overloading of the specific operators.

1. *Overloading of operators test_big_number.py* - we've tested the following operators: *__repr__* - the printable representation of the big number, *__add__* - the addition between two big numbers, *__sub__* - the substraction of two big numbers, *__mul__* - the multiplication between two big numbers, *__truediv__* - division of two big numbers, *__xor__* - the operator used for power, *__invert__* - the operator representing the square root operations. By overloading these operators, we call the corresponding method from the *Operations* class and we return a new *BigNumber* object that stores the computation as value. For that, we've needed to use mocking in order to receive the *Operations* objects.

```
def test_add(self):
    with unittest.mock.patch
        ('core.operations.Operations.add_numbers') as op:
            nr1 = BigNumber('3')
            nr2 = BigNumber('2')
            nr1.__add__(nr2)
            op.assert_called_once_with('3', '2')
```

Figure 5. Testing the '+' overloading inside the BigNumber class

2. Computations results
   (a) Addition *test_big_number_add.py* between:
      i. Big positive numbers

```
self.big_positive_number_1 = BigNumber(
    number="1988431321213931232892 3")

self.big_positive_number_2 = BigNumber(
    number="29201938109201298310982301928 30")

self.result_object_1 = BigNumber(
    "29201938308044430432375425217 53")

self.assertEqual((self.big_positive_number_1 +
    self.big_positive_number_2), self.result_object_1)
```

      ii. Small positive numbers
      iii. Very big positive numbers
      iv. A negative and a positive number

v. Items with invalid format

```
self.wrong_char_number_1 = BigNumber("dq3282023ds")
self.positive_number_1 = BigNumber("123903123312")
with self.assertRaises(ValueError):
    self.wrong_char_number_1 + self.positive_number_1
```

(b) Difference *test_big_number_difference.py* between:

i. Big positive numbers

ii. Big positive numbers with negative result

```
self.big_positive_number_1 = BigNumber(
    "1988431321213931232892 3")
self.big_positive_number_2 = BigNumber(
    "29201938109201298310982301 92830")

self.result_difference_1 = "29201937910358166189589178 63907"

self.assertEqual(
    (self.big_positive_number_2 - self.big_positive_number_1).number,
    self.result_difference_1)
```

iii. Small positive numbers

iv. Small positive numbers with negative result

v. Very big positive numbers

vi. Very big positive numbers with negative result

vii. Negative numbers

viii. Items with invalid format

(c) Division *test_big_number_divide.py* between:

i. Small positive numbers

ii. Big positive numbers

iii. Very big positive numbers

iv. Zero division

```
self.big_positive_number_1 = BigNumber(
    "19884313212139312328923")
with self.assertRaises(ValueError):
    self.big_positive_number_1 / BigNumber('0')
```

v. Negative and positive numbers

vi. Items with invalid format

(d) Multiplication *test_big_number_multiply.py* between:

i. Big positive numbers

ii. Small positive numbers

iii. Very big positive numbers

```
self.very_big_positive_number_1 = BigNumber("123" * 100)
self.very_big_positive_number_2 = BigNumber("256" * 101)
self.result_object_5 = BigNumber("31551070590109629148668187707226746265785304

self.assertEqual(
    (self.very_big_positive_number_1 * self.very_big_positive_number_2),
    self.result_object_5)
```

iv. Negative numbers
v. Items with invalid format

(e) Power *test_big_number_power.py* between:

i. Positive numbers

```
self.positive_number_1 = BigNumber("198")
self.positive_number_2 = BigNumber("10")
self.result_object_1 = BigNumber("926087244809015797777024")

self.assertEqual(
    (self.positive_number_1 ^ self.positive_number_2),
    self.result_object_1)
```

ii. Negative numbers
iii. Items with invalid format

(f) Squared root *test_big_number_sqrt.py* for:

i. Big positive numbers

```
self.big_positive_number_1 = BigNumber(
    "19884313212139312328923")
self.result_object_1 = BigNumber("141011748489")

self.assertEqual(
    (~self.big_positive_number_1), self.result_object_1)
```

ii. Small positive numbers
iii. Very big positive numbers
iv. Negative numbers
v. Items with invalid format

Operations class

Here will be defined a test object of type *operations.Operations()* in order to directly compute the results using numbers, instead of BigNumbers objects.

The same types of test cases are applied, each operation tested into a different script: *test_add_numbers.py, test_difference_numbers.py, test_divide_numbers.py, test_multiply_numbers.py, test_pow_numbers.py, test_sqrt_numbers.py.*

Example:

```
self.big_positive_number_3 = "1000000000000000000000000"
self.test_object = operations.Operations()

self.assertEqual(
    self.test_object.sqrt_numbers(first_number=self.big_positive_number_3),
    self.result_sqrt_3)
```

Parser class

We've performed testing on this class for evaluating how the expressions are parsed and represented on a syntax tree.

We've tested the following cases:

1. Parsing an invalid expression

```
self.test_object = expression_parser.Parser()
self.wrong_expression = "a+(b*c"
self.values2 = {'a': '2', 'b': '3', 'c': '4'}

with self.assertRaises(Exception):
    self.test_object.parse_expression(formula=self.wrong_expression,
    values=self.values2)
```

2. Raising error if (some) varibles values are missing

```
self.test_object = expression_parser.Parser()
self.expression2 = "a*b+c"
        self.less_values = {'a': '2', 'c': '3'}

  with self.assertRaises(Exception):
            self.test_object.parse_expression(formula=self.expression2,
            values=self.less_values)
```

3. Parsing a regular expression

```
self.expression = "a+(b*a)"
self.values = {'a': '1', 'b': '2'}
with unittest.mock.patch
    ('core.expression_parser.Parser.parse_expression') as parse:
        self.test_object.parse_expression(formula=self.expression,
        values=self.values)
        parse.assert_called_once_with(formula=self.expression,
        values=self.values)
```

Computation class

These tests evaluate the computations of results when traversing the syntax tree.

We've tested the following cases:

- A regular expression
- A complex expression

```
self.big_expression = "a+(b*c*((d^2)/a)-~(e))"
self.values2 = {'a': '2', 'b': '3', 'c': '4', 'd': '8', 'e': '64'}
self.result2 = '378'
self.test_object = compute.Computation()

self.assertEqual(self.test_object.get_result(formula=self.big_numbers,
    parameters=self.values3)[0].number, self.result3)
```

- An invalid expression
- Raising error if (some) varibles values are missing

5.2 Use of assertions

In this section is shown how assertions are used for each class as preconditions, postconditions and always true conditions inside a function. Every type of condition enumerated before is used for each function in each class.

- Operations class
    1. Preconditions: We check if the number is negative, if the number has not only digits and if a number start with 0 and is different from 0.
    2. Always true conditions: We check if numbers are as type string and if are different than None.
    3. Postconditions: We check if the result given by the operations are equal with the one given by the python interpreter.

```
def assert_postconditions(result, operation, number_1, number_2):
    if operation == '/':
        assert str(eval(number_1 + "//" + number_2)) == result,
        "different result"
    elif operation == "^":
        assert str(eval("pow(" + number_1 + ',' + number_2 + ")")) == result,
        "different result"
    elif operation == "~":
        assert str(int(eval("sqrt(" + number_1 + ")"))) == result,
        "different result"
    else:
        assert str(eval(number_1 + operation + number_2)) == result,
        "different result"
```

- BigNumber class

  1. Preconditions: We check if the number obtained from the BigNumber object is negative, if it has not only digits and if it start with 0 and is different from 0.

  ```
  def assert_preconditions(number):
      assert number.count('-') == 0,
      "the number is negative"
      assert number.isnumeric() is True or number == "0",
      "the number is not numeric"
      assert (number[0] == '0' and len(number) == 1) or number[0] != '0',
      "The number can't start with 0"
  ```

  2. Always true conditions: We check if numbers given by the BigNumber objects are as type string and if are different than None. We also check if the result of the operations between 2 BigNumber objects is also a BigNumber object and different from None.
  3. Postconditions: We check if the result given by the overridden arithmetic operators are equal with the one given by the python interpreter.

- Parser class

  1. Preconditions: We check if the formula length is greater than 0 and if the number of the opened parentheses is equal with the number of the closed parentheses.
  2. Always true conditions: We check if the formula is as type as string and if it is not None. We also check if the parameters are as type as dict and also if the parameters are not None.

  ```
  def assert_invariant(formula, parameters, parameter):
      assert type(formula) == str,
      "different type for expression"
      assert formula is not None,
      "expression is None"

      assert type(parameters) == dict,
      "different type for expression values"
      assert parameters is not None,
      "no values for expression"
      for keys, values in parameters.items():
          assert type(values) == str,
          'Different type for parameters'
      if parameter:
          assert parameter in parameters.keys(),
          "no value for parameter " + parameter
  ```

- Computation class

  1. Preconditions: We check if the formula length is greater than 0 and if the number of the opened parentheses is equal with the number of the closed parentheses.

     ```
     def assert_preconditions(formula):
         assert len(formula) > 0,
         "empty expression"
         assert formula.count('(') == formula.count(')'),
         "incorrect parenthesis"
     ```

  2. Always true conditions: We check if the formula is as type as string and if it is not None. We also check if the parameters are as type as dict and also if the parameters are not None.
  3. Postconditions: We check if at each step of computing the expression, the result given by the overridden arithmetic operators are equal with the one given by the python interpreter.

6. Conclusions

Although the requirements of the project may seem trivial at the first sight, implementing them in a flawless manner is no easy task, that can be completed thoroughly only by making use of software quality techniques, such as a well defined object oriented architecture and extensive testing. This way we can make ourselves sure that each module is correct with respect to the requirements, and also the modules integrate between themselves without unexpected issues. A well written program should always return the correct answer for a valid input, while being able to distinguish an invalid input sequence without crashing, alerting the end user about it.

7. Future work

For the future are some operations that can be improved like division and square root. We need to find more efficient algorithms to minimize execution time.

As another idea for the future work the code of this project can be reused as a module for a bigger project because it is tested well and has a clean code.

References

[1] Lect. dr. Vlad Rădulescu - Materials of Sofware Quality

[2] Proposed subject https://profs.info.uaic.ro/~rvlad/lab/cs/proiect-cs.html

[3] Source code https://github.com/RaresM123/CSS_PROJECT/tree/phase_2