Big Number Calculator

Crețu Bogdan-Antonio Minuț Mihai-Dimitrie

May 31, 2022

Contents

1 Introduction **3**

2 Implementation **3**
- 2.1 Mihai's Implementation . . . 3
  - 2.1.1 Architecture . . . 3
  - 2.1.2 Big Numbers . . . 3
  - 2.1.3 Post fixed expressions . . . 4
  - 2.1.4 User Dialog . . . 5
- 2.2 Bogdan's Implementation . . . 5
  - 2.2.1 Architecture . . . 5
  - 2.2.2 XML parsing . . . 5
  - 2.2.3 Expression parsing . . . 5

3 Unit Testing **5**
- 3.1 Mihai's Unit Testing . . . 5
  - 3.1.1 Post fixed expressions . . . 5
  - 3.1.2 User Dialog . . . 6
- 3.2 Bogdan's Unit Testing . . . 6
  - 3.2.1 Big Number . . . 6
  - 3.2.2 User dialog . . . 6

4 Assertion Testing **6**
- 4.1 Mihai's Assertion Testing . . . 6
  - 4.1.1 Post fixed expressions . . . 6
  - 4.1.2 User Dialog . . . 7
- 4.2 Bogdan's Assertion Testing . . . 7
  - 4.2.1 Big Number . . . 7
  - 4.2.2 User dialog . . . 7

5 Conclusions **7**

1 Introduction

This document represents the documentation for the Big Number Calculator application's code built for the Quality of Software Systems.

The content of the document can be broken down into: table of contents, implementation phase, unit testing phase and assertion testing phase; each separated by each contributors category for clearance of work attribution.

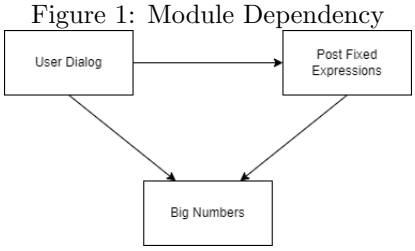
2 Implementation

2.1 Mihai's Implementation

2.1.1 Architecture

The application architecture is built upon 3 layers that have a simple dependency schema (1): big numbers, post fixed expressions and user dialog.

Figure 1: Module Dependency



The figure 1 describes 3 modules:

1. User Dialog, which depends on Post Fixed Expressions and Big Numbers, and that handles the user input and maps it to application logic
2. Post Fixed Expressions, which depends on Big Numbers, and that handles the text parsing and validation of big number arithmetical expressions. It is responsible for checking the syntax and the semantic meaning of an expression, coming from/going to different input/output destinations, but also solving and preparing solving history output to be displayed to the user.
3. Big Numbers, which is a base representation class within the application; it is used to interpret big numbers coming from positive numeric strings and process different operations with them: addition, multiplication, division etc..

2.1.2 Big Numbers

The Big Numbers module contains only the class "BigNumber" which implements the concept of a big number by storing the big number digits as integers within an array object.

The string numbers given as input to the constructor should always be positive and contain only digits. Within the constructor the string number is parsed and checked by various criteria in order to validate the number. The constructor also requires a parameter which specifies the maximum allowed number size, which triggers a specialised error when being overstepped in operations or initialisation.

The list of possible operations with a big number include:

1. Equality between 2 big numbers
2. Lower or equal between 2 big numbers
3. Lower than between 2 big numbers
4. Hashing is disabled, and should not be used
5. Representative and string conversion methods are updated to match the class behaviour

6. Digit index read and write access is supported
7. Arithmetic addition between two big numbers is implemented as vector addition with carryover
8. Arithmetic multiplication between two big numbers is implemented as multiple additions
9. Arithmetic substraction between two big numbers is implemented as vector substraction with carryover
10. Integer division is supported between two big numbers and is implemented by repeated substraction
11. Arithmetic pow can be used between two big numbers and is implemented by repeated multiplication
12. The root operation between two big numbers is implemented, only for the square root, by applying multiple divisions, additions and substractions.

The list of possible custom errors that can be thrown while working with the class include:

1. When the result of an operation becomes a negative number
2. When an illegal symbol is found within the init string
3. When the big number string starts with a zero
4. When the resulting number size is bigger then the restricted size
5. When a number is divided by zero
6. When the root power is not square root

2.1.3 Post fixed expressions

The post fixed expressions module is composed of only one class, the "PostfixedExpression" class, which is responsible of holding the validation, postfix transformation and parsing methods of an infix arithmetic expression received as input.

The class variables include the **precendence** which contains a hashmap of all the possible operators as keys that have mapped their precedence order, which represents in which order the operations are interpreted. Another map with string operators mapped to all "BigNumber" arithmetic operators is included within the class variables.

The class can be configured by outside factors to either show less or more console output or increase or decrease the size of the big numbers that will be allowed.

The methods of the main class can be listed as:

1. solve - which iterates through each solving step and saves the output of each atomic operation performed
2. show_solving_history - which uses the previously saved output to print to the console each step of the solution, if the verbosity is set to full, or just the expression and its result if it is set to false.
3. init_expression - initialises the post fixed expression stack by using the syntax parsing operator then tries to restore the expression to an infixed one so all the syntax errors can be found
4. restore_post_fixed_expression - pops all the items from the remaining post fixed expression stack to construct the remaining expression string
5. build_post_fixed_expression - parses and validates the infix expression string along with building the post fixed expression stack by using the precedence value of the operators and transforming string digit sequences into big numbers.

The custom errors that can appear while using the "PostfixedExpression" class are:

1. Misplaced Symbol - a symbol appears within the given input expression at an invalid position
2. Illegal Symbol - the expression contains symbols which are not allowed
3. Additional Closing Bracket - the expression contains too many closing brackets

2.1.4 User Dialog

The user dialog is a module with mostly static methods that are mappings of text given user inputs within the main application loop.

The user input events that are implemented through the user dialog module are:

1. A welcome message when the application starts
2. A help command which prints all the available commands and their short introduction
3. A method of showing the currently loaded big number expression
4. A method for letting the user import/export a expression to a XML file to the OS file system
5. A method for letting the user change the current expression through console input
6. A method used to modify the verbosity
7. A method used to modify the number allowed size
8. A method to display the current's expression solution and its steps, depending on the verbosity

2.2 Bogdan's Implementation

2.2.1 Architecture

The architecture was discussed with Mihai as a 3 main components application. The implementation of each module being parallel.

2.2.2 XML parsing

The main component I worked with was parsing of XML files and writing to xml files. The application is able to read an expression form a XML file given in the simple format of symbols, operators, and numbers, with the result and the expression itself also being entities.

The XML paring reads each component and transforms it's content into the corresponding part of a string-based expression that is afterwards turned into a post-fixed expression

2.2.3 Expression parsing

On the expression parsing (and conversion to post-fixed expression) I worked with Mihai, I created an initial post-fixed expression parser with some basic operators, and Mihai reiterated it to fit the implementation of the Big Number and to do some optimizations and a bug-fix.

3 Unit Testing

3.1 Mihai's Unit Testing

3.1.1 Post fixed expressions

Each method of the post fixed expression module was tested with multiple unit tests with the goal of testing the normal use cases and also the niche use cases that could cause problems.

The method for building post fixed stack expressions was tested with a valid expression, a null one, a syntactically incorrect one and an illegal one given as input. The setup for all of those tests was an initialised expression.

The method for solving the expression was set up similarly and the tests featured: the normal behaviour case, the incorrect syntax within the expression case and the invalid resulting number limit size expression case.

The process of restoring a post fixed stack to an infix string expression was tested to: show correct input, throw misplaced symbol errors, throw big number errors, to throw exceptions when the expressions are empty and to correctly restore the expression even if initially it does not contain brackets.

The methods of importing and exporting big number expressions through XML files were tested using mocking on the IO methods and by validating correct inputs and outputs with the correctly matching results and as well tested with failing inputs.

The method that printed the solving history to the console was tested by mocking the standard output channel builtin handler and matching correct and incorrectly given expressions with correct outputs or error messages.

The example given within the module was tested to run independent to the origin directory and as well to have the desired behaviour.

The constructor was tested to throw errors in the edge cases when an incorrect input was provided and to as well behave correctly in the desired use cases.

3.1.2 User Dialog

Each method of the user dialog module was unit tested by having the standard input/output and the post fixed expression class mocked in order to observe that it correctly handles the user input and output.

The help method was tested to correctly fill in parameters given by the post fixed expression.

The methods that changed the verbosity and maximum allowed number sized were tested for testing the input validity and correctly assigning the values, if correct, to the post fixed expressions module.

The method that showed the solving steps for the expression was tested against invalid expressions, unsolved or empty expressions and as well with normal cases where the correct solving history was matched accordingly.

The expression given user input functionality was tested against empty expressions, expressions filled with invalid symbols, invalid operator syntax order or invalid semantic interpretation.

3.2 Bogdan's Unit Testing

3.2.1 Big Number

For the unit testing phase we swapped up the components we work on, I covered the Big Number class implementation, My main target was to ensure that each method of the class would be able to follow the intended use-case, and to check that negative cases would result in proper errors being returned, and stop the method execution.

3.2.2 User dialog

For the User dialog the methods were shared arbitrarily between team members, my components are: Show Current expression, Change current expression XML, Interactive Menu, Modify Verbosity, Solve Current Expression.

For the user dialog section the approach was similar to the big numbers section, the unit tests covering positive use cases, and some failure scenarios.

4 Assertion Testing

4.1 Mihai's Assertion Testing

4.1.1 Post fixed expressions

Each method of the post fixed expression module was tested with multiple insertion of assertion with the goal of validating the invariants and the preconditions and post conditions necessary for the correct behaviour.

The method for building post fixed stack expressions was tested to assert complete parsing of the characters at the end and correct operator mapping and precedence during the parsing. The processed numbers were also tested to assert correct digit representation before converting to big number.

The method for solving the expression was set up to assert that operators were of the correct string type while the numbers were all of type "BigNumber". The final output was tested to contain only one item, it being the expression big number result.

The process of restoring a post fixed stack to an infix string expression was tested to assert that operators were applied only between two "BigNumber" instance types and that the ending operation stack was composed of only one item, the result of the interpreted expression.

The methods of importing and exporting big number expressions through XML files were tested to assert that the giving output/input does not contain illegally placed symbols after the preprocessing and that the resulting file/expression is valid after parsing.

The constructor was tested to assert that the processed expression is bigger than length 0 and also that all the non operators from the postfixed expression stack are of type "BigNumber".

4.1.2 User Dialog

Each method of the user dialog module was tested to assert the correct processing of the user input and the correct course of actions when mapped to the previously checked input.

4.2 Bogdan's Assertion Testing

4.2.1 Big Number

For assertion testing the same strategy from the unit testing phase was used, I searched for conditions that should be respected at the big number level, and verified that variables don't reach unintended values.

4.2.2 User dialog

For the user dialog the same distribution was used, my tested methods were: Show Current expression, Change current expression XML, Interactive Menu, Modify Verbosity, Solve Current Expression.

Most of the time used for the user dialog was spent considering invariants and conditions that should be asserted at this level of the application. In the end I have written very few assertions on this component.

5 Conclusions

We found out that designing test cases and test scenarios is quite time consuming, but we managed to get a good application running and although our testing revealed some problems, we re happy with the results and the process we followed during the project phases.

We managed to organize quickly, and using a combination of pair and parallel work we managed to achieve good results in each part of the project.