DOCUMENTATION

ASSIGNMENT 1

STUDENT NAME: BÁLINT ROLAND
GROUP: 30424

TABLE OF CONTENTS

1. Objective

The main objective of this project is nothing less than implementing a Polynomial Calculator.

The calculator will work on one or two polynomials (depending on the type of the operation), which will be inserted by the user through an easy-to-use graphical interface. The possible operations are:

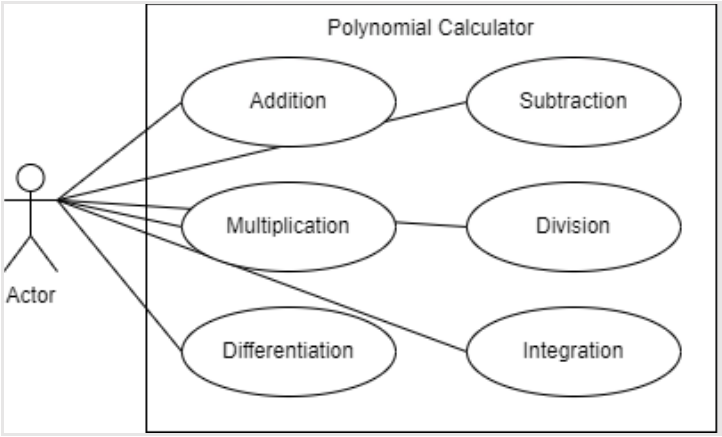
- Addition
- Subtraction
- Multiplication
- Division
- Differentiation
- Integration

Besides reaching a correct implementation, using the correct design patterns, and ensuring the possibility of user-program interaction, some unit tests will be written to check if the calculator behaves the way it was intended. The code will be followed by a rigorous documentation (this file).

2. Analysis, modeling, scenes, use cases

During this project, I will focus on the functional requirements since the problem statement lacks information about nonfunctional ones. Of course, I will thrive for optimality, readability of the code, to make future development or evaluation easier, but my main goal is to achieve a correct implementation of the polynomial operations. As for the user, my calculator is in fact a simple tool, capable of reducing human efforts when working on this branch of mathematics.

The following use case diagram shows what my product can do, or in other words, what can the user do with it:



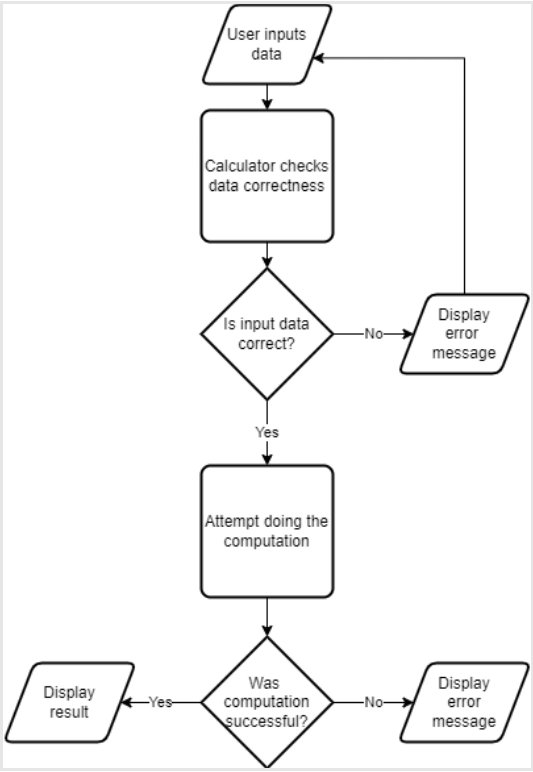
Use Case Diagram of the Polynomial Calculator

As it was mentioned previously, and as it can be seen on the attached diagram, the actor (user) can use the Polynomial Calculator to perform six different tasks: addition, subtraction, multiplication, division, differentiation, and integration. In order to do such computation, it is more than obvious that we need some data to work with. In the case of addition, subtraction, multiplication and division, two operands (polynomials) are required, while differentiation and integration work on a single polynomial.

The information fetching process is simple: the user inputs the polynomials as text, using their keyboard or the buttons on the graphical interface, in a given format, resembling the mathematical notations. This data will be read and evaluated. There is a high chance of inserting data which is incomprehensible for the calculator. In such a

case, an error message will be displayed, and the user is requested to correct their input. Other alternative scenarios may occur if the operation causes negative or positive overflow. In fact, any of these operations except division can result in overflow. In case of the last, we may be faced with division with zero, which should be treated as a special case, as well.

This simple process is illustrated in the flow chart below.



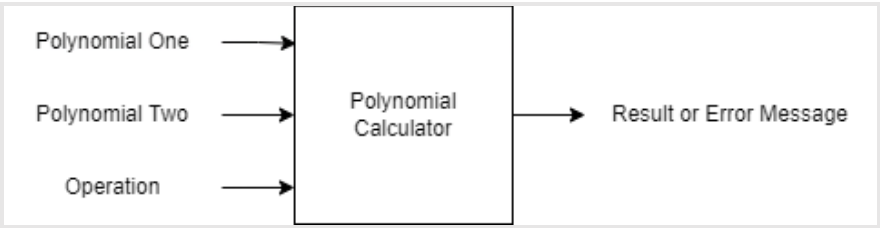
Flow Chart Representing All Possible Scenarios

Note that, in the UML flow chart I did not include the starting and terminating nodes. This is because they are independent of the flow of data and computation: the application is running from the moment it is launched and terminates when it is closed using the "X" button. The correctness checking and computation processes are fired by the user when he/ she presses the button with the corresponding operation name on them. These verifications, from the user perspective, happen at the same time. In reality, the code follows the direction described in the diagram.

The operations happen instantly, since the size of the processed data is small. The steps of computation are the same for every operation (except the algorithms, of course). If a one-operand operation is desired, only one input will be verified (more details on this later).

3. Design

As a conclusion to everything written until now, and before digging deeper into implementation, an abstract view, or "black box" of the system would look as follows:

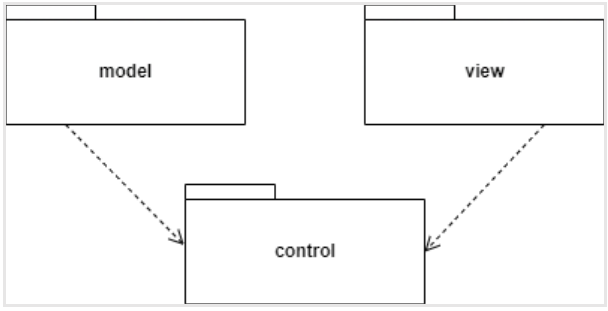


Overall System Design

As a good programming practice, it is recommended to think before coding. When using object-oriented languages, it is especially important to divide the problem on smaller partitions, possibly on objects which resemble real life entities, or at least tie together things related to a single theme.

In order to keep things organized, nice and elegant, I chose to use a design pattern I am somewhat familiar with: the Model-View-Controller architecture. This implies that I have to divide the problem in three big parts. The first one, and arguably the most important, is the Model, which will contain the logic, the operations themselves, the data types, everything that is needed to perform the tasks. The Model in itself should provide a working solution of the problem, but without an interface. The View will contain everything related the Graphical User Interface. This independently is very nice, but lacks every kind of functionality. The Controller will tie these two together, resulting in a functional program.

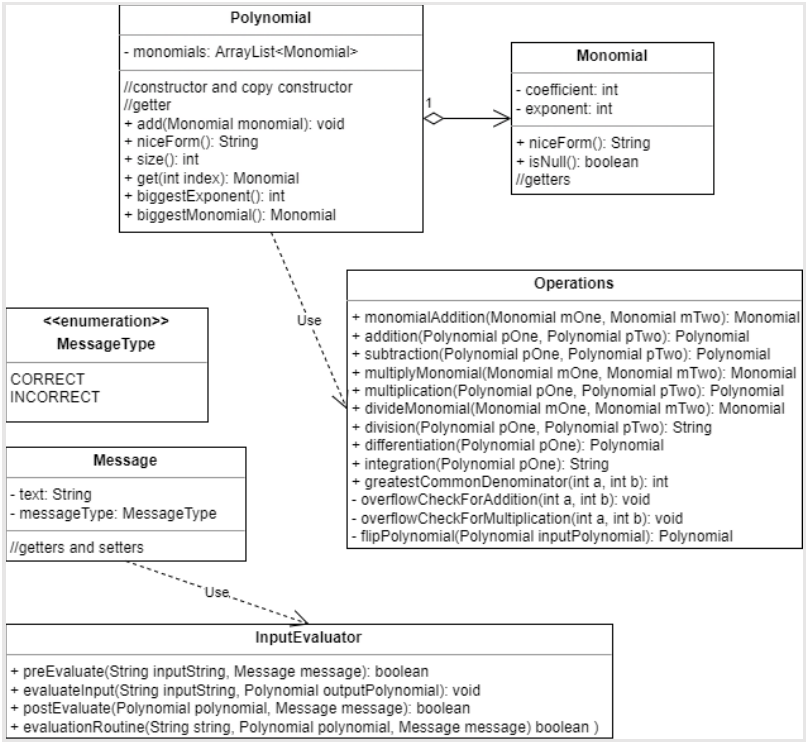
The very first thing that I did in order to follow the MVC architecture was to divide my program into three packages, each resembling (and in future, containing the implementation of) one of these parts.



Package Diagram

After this, I started working on the content of each package. First, I wrote the model and the view in parallel, just to avoid boredom. Even though I developed these two together, I will start with the model, since it is more logical to describe first the solutions I came up with.

The classes of the Model look like this:



Class Diagram - Classes Inside Package "model"

Let's start with the very beginning. To work with polynomials, I needed to define the Polynomial class. What is a polynomial? A sum of monomials. So, first I should define the Monomial class. A monomial is fully defined by its sign, exponent, and coefficient- more about this in the implementation section. For now, just focus on fully understanding the class diagram. To keep it nice and fancy, I used the data type ArrayList to realize this aggregation relation. This data structure comes with many helpful facilities, which will come in handy later.

The InputEvaluator class contains only static methods which relate to the evaluation of the inputs. Some of its methods, through the Control, will use the class Message, as a message transmitter between classes and methods; meanwhile this Message class will get use of the MessageType enumeration.

Every polynomial (and monomial) operation can be found in the Operations class. These are static methods as well, and take as parameters two or one polynomial(s). But again, more on this later. For now, since this section is dedicated for used algorithms as well as design choices, I will briefly describe the mathematical background needed to write this code, and supposedly to understand it, for each operation.

a. Addition

This, first operation is straight-forward. It is important to remember that our problem is reduced to polynomials with one variable (and integer coefficients, and positive and integer exponents). In this particular case, all one needs to do is to add the coefficients of the polynomials with the same power.

b. Subtraction

If addition is implemented correctly, subtraction is elementary. Why so? There is a practice, especially in hardware, that subtraction is perceived as negating one operand and adding it to the other, and this case is no different. This is especially good for us, since it will enhance our code quality since we reuse already written methods.

c. Multiplication

Multiplication is where it starts to get a little bit harder. Multiplying two polynomials is exactly the same as "opening some parenthesis" and simplifying the result. This means multiplying each monomial on one side with each monomial on the other side. The operation resembles a Cartesian product. The simplification step consists of adding up elements with same exponent. This will be important, as you will see.

d. Division

Division was, by far, the hardest operation. For this, I used the method of "Long Division". Its steps may lead to confusion, and they can be interpreted in many ways; I will try to describe it in my own words. First, we look at the biggest exponent monomials of both operands (dividend and divider). We will divide these, getting as result a monomial which shows "how many times is the divider present in the dividend " (just like quotients in any other context). We will take this result, multiply it with the divider, switch its sign (multiply by -1, flip it, call it as you wish) and add it to the dividend. Repeat the process on the remainder, as long as it (the current remainder) becomes of a lesser order than the divider (we can no longer continue the dividing process).

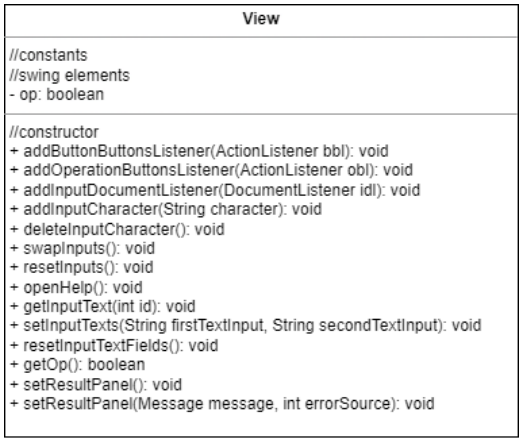
e. Differentiation

The differentiation of polynomials is the most primitive form of differentiation. We have to apply only one algorithm, which is: multiply the coefficient with the exponent, and decrement the exponent, for each monomial. The order is important, as, for example $x^3dx = 3x^2$ and not $2x^2$.

f. Integration

Integration is the opposite of differentiation. In polynomial algebra, it means incrementing the exponential and dividing the result by the same number as the exponential (again, in this order). Since we talk about integration without limits, we have to add the +C at the end, meaning the answer is a set actually, not a single number.

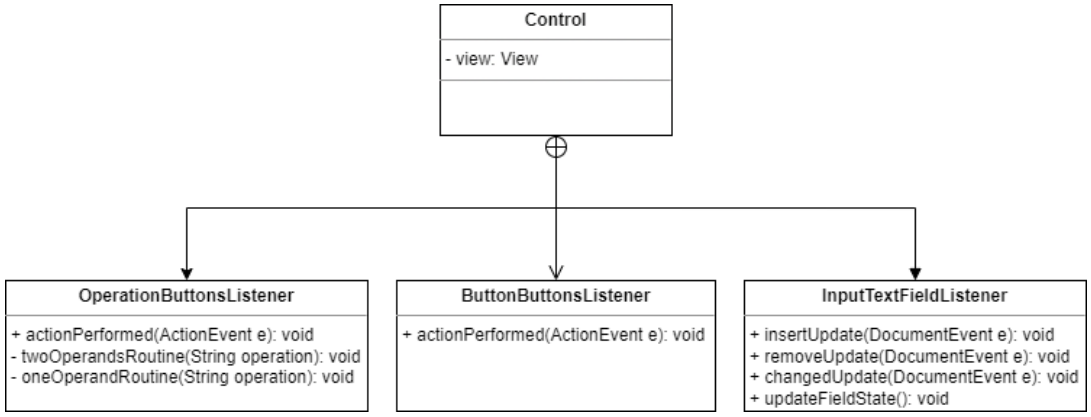
Continuing with the next package and its classes, the package view contains one single class, holding the same name (but with uppercase first letter, following the java language conventions, of course).



Class Diagram - Classes Inside Package "view"

This class contains the various java swing components making it possible to create a graphical user interface. Aside of these, there are some constants defined, such as the name of the buttons (and operations), or the text of the help pop-up. This class is home of various graphics related methods.

And, the class which brings all this alive: Control. Being the singleton class of package “control”, it is responsible for tying together the elements of the view and the model.



Class Diagram - Classes Inside Package "control"

As depicted above, the class Control has three inner classes: OperationButtonsListener, ButtonButtonsListener and InputTextFieldListener. This makes things more organized, and passes the responsibility of implementing event related methods from the main class to its descendants. Contrary to what can be seen in usual MVC projects, there is no instance of Model present is Control class, this is firstly because there is no such class (the model part is divided in several classes), and secondly because the one class we need, the Operations class, has only static methods, aka they can be invoked without an instance. So, even if on the visualization it is not so obvious, the control class does use the operations defined in model (otherwise there would be no application at all).

One last class, also worth mentioning is the one where the testing is done, called OperationsTest. I will provide details later, at the Results section, for this moment I will only attach its diagram.

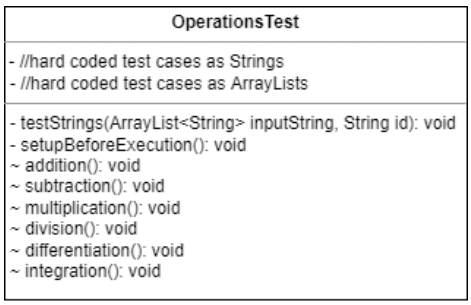


Diagram of OperationsTest

4. Implementation

All right, finally, here come the implementation details. To keep some kind of consistency, I will go through the classes in the same order as in the previous section

a. Classes of package "model"

Monomials: the basis of everything in this project. They can be fully described by three characteristics: exponent, coefficient, sign. Since there is no unsigned data type in Java, and since keeping it separately seemed to me as wasted effort, I kept the sign and the coefficient together. So, my Monomial class had the following attributes:

```
private final int coefficient;    //integer, containing sign as well
private final int exponent;       //non-negative and integer, as specified in problem statement
```

These fields are final, since during the implementation I never encountered a case in which I should have modified a monomial (in situations where this was a possible solution, creating a new monomial/ polynomial seemed easier to make and to follow).

During the process of code writing, I often used the tactic of splitting big tasks into smaller ones, and in many cases this meant that the operations of Polynomials were first done on Monomials and then used in the other as needed. The first example of this is the niceForm method, which delivers a user-friendly form of a polynomial. Due to the structure of a polynomial, repeated printing of monomials results in printing the polynomial itself. Of course, in theory it is very easy to print a monomial, but in practice, there are a lot of special cases, such as: exponent zero means no "x" printed, exponent one should not be specified at all, positive sign should be printed, of course, but with one exception- when it is at the beginning of the polynomial. So yeah, for the user, the polynomial is the nice form, but for me, as a developer and programmer, it is nothing more than an exponent and a coefficient.

Also in the polynomial class, I defined some methods which I like to refer to as "helper methods". Their existence isn't essential, but I think they are a nice addition, and they make the code easier to understand. Such methods are: size, biggestExponent, biggestMonomial, get. The add method deserves a special place, since it has a special mechanism which I had to add to it to make the division work (and ultimately, to make the code absolutely correct), by which I mean that it does not add elements (monomials) to the polynomials if they are null (their coefficient is zero).

```
//adds a monomial to the list of monomials
public void add(Monomial monomial){
    if (monomial.isNull())
        return;
    monomials.add(monomial);
}
```

This means that the there is no internal representation of a zero-polynom, a zero is equal to an empty polynom (its monomial arraylist is empty). This case is dealt with in the niceForm method.

Now, I could write essays about the Operations class, since its development took ages, and there are a lot of small details to cover, but I will try to highlight only some special aspects I met, which took some time to figure out and significantly slowed down the implementation process.

The first such thing is, as obvious as it may sound, that working with objects as parameters will change their state out of the method as well. This was avoidable up to a point. I reached to a significant depth of the problem when I realized that my subtract method will permanently negate its second operand. This caused problems in the long division algorithm, since the same object was re-used many times. The simplest and most elegant solution to this was creating a flip method:

```
//returns the "reverse" of inputPolynomial - flips each monomial's sign
private static Polynomial flipPolynomial(Polynomial inputPolynomial) throws OverflowException{
    Polynomial invertedPolynomial = new Polynomial();
    ArrayList<Monomial> inputMonomials = inputPolynomial.getMonomials();
    for(Monomial m : inputMonomials){

        overflowCheckForAddition(m.getCoefficient(), b: -1);

        invertedPolynomial.add(new Monomial( coefficient: m.getCoefficient()*-1,m.getExponent()));
    }
    return invertedPolynomial;
}
```

Remember when I said that the add function is extremely important? Well, I felt this when I tried to implement the multiplication method. For me, during the whole program, a valid polynomial meant a polynomial which is fully simplified, with its monomials having their exponents in (strictly) decreasing order, and having no null elements. The last aspect is covered by the add function, but how come the order of monomials is secured?

Well, firstly the user is enforce to enter the data this way. So, the input comes in this form, and it is evaluated in little chunks, following the correct order. This order is maintained since I used list as data structure.

As mentioned many times, the implementation of division was the biggest trauma of this project. I wrote the algorithm twice, and it kept entering an infinite loop, for a reason I could not find. Finally, after going through the used methods (subtraction, multiplication) and after a good nights sleep, I managed to nail it, on the third attempt. Also important to note that this method, as well as the integration method, return Strings, since the result cannot be displayed as a polynomial, but rather as two polynomials (in the case of division, the quotent and the remainder) or as a polynomial and some extra logic (in the case of integration, sometimes we need a denominator, and some paranthesis for correct, symbolic representation).

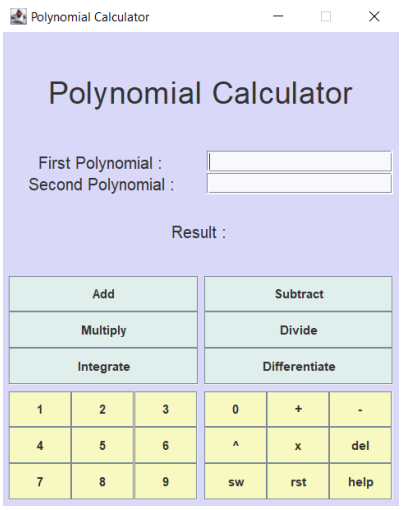
Input evaluation is done on three levels. On the first level, the input is checked, using regex, if contains only legal characters.

```
//check if contains anything except characters needed to describe a polynomial, using regex
if (!inputString.matches( regex: "^[0-9x^+-]*$")){
    System.out.println("Illegal character in input.");
    message.setText("Illegal character in input.");
    return false;
}
```

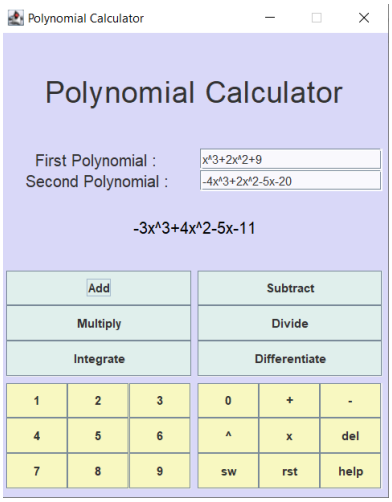
Of course, other evaluations are ran as well, but I will not enter in too many details (the code is pretty self-explanatory). The main idea is that after the semantical correctness test is checked, the String is converted into a polynomial. Only after this, the post-evaluation phase is entered, where the exponent of the monomials are checked (only strictly decreasing order is accepted).

b. Classes of package “view”

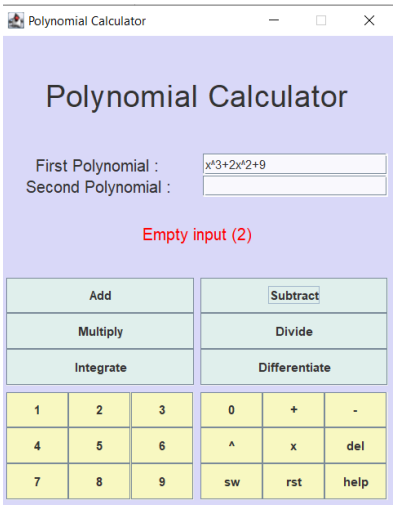
There is just one class here, View, which contains every swing element. I tried to imitate the GUI presented in the support presentation, with some minor changes to it.



Look upon startup



Display the Result



Display the Error

I think it looks kinda cute.

c. Classes related to package “control”

Out of other aspects, the most important one related to control class is the implementation of actionPerformed and insertUpdate, removeUpdate, changedUpdate methods. The user interaction is divided in three parts: the “button buttons”, aka the lower 18 (yellow) buttons, the “operation buttons” (the green ones) and the two text fields. To respect this division, which was first introduced by the structure of the View class, each part has its own class, describing the responses to user interaction. For example, the lower buttons can be used instead of the keyboard- it has all the buttons/ characters necessary to input the data, there is a delete button, and other, extra features, such as swap, clear, help. The upper buttons trigger an operation. The text fields make it possible to enter data. Since the user can write from keyboard, and there is not “enter input” button, I needed to update the internal state of the computer in synch with what happens on-screen. This is how I got introduced to document listeners, which is a really cool concept, but it somehow interfered with my switch and reset functions. So, I did the “lumberjack method”, and came up with the simplest (and not so elegant) solution of introducing a Boolean variable which is set in case of switching or clearing. When this Boolean value is true, the document listener is “disabled”.

5. Results

Finally, with a lot of work, headaches, and luck I managed to implement all operations correctly (at least I think so). This statement is preceded by a lot of testing, debugging, looking for special, corner cases. Out of the target objectives I reached all, even if the code itself is not perfect, and some solutions are not the best.

First of all, I learned that there is no such thing as flawless code, at least not on my level, yet. Every time I looked, I found something to improve, to waste my time on. And sometimes I fixed the code so much, that it broke. In these situations, it was extremely helpful that I had backups on GitLab.

Aside of git, I used many technologies for the first time. For example, I never tested with Junit before. The hardest part of it was setting it up, I spent a whole afternoon/night looking for fixes, because the dependencies did not work. In the final, I had to remake the project, but not as a Maven project, but as a simple one, and add the test package manually.

6. Conclusions

The testing part definitely can be enrolled in the future developments category, since I am very new to it, and had no time/ resources to get deeper into it. So, what I did is what I call a “semi-automated” test: I hard coded some operands, the expected values, and I ran the tests on them. It would have been more elegant to check on different corner cases on different operations, to generate random test values, etc. Junit seems like a powerful tool, but I do not know how to use it, really.

The GUI is not perfect, it does the job, but it is also limited due to my lack of front-end knowledge. Even like this, it took ages to reach an acceptable look. I could have made some better design choices at times, and less restrictions for the user input would have been better, as well.

But, to end this document on a high note, I have to admit that I am satisfied with the outcome of this project, and I feel like I learned a decent amount of new things

7. Bibliography

Course materials

https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_1.pdf
https://dsrl.eu/courses/pt/materials/A1_Support_Presentation.pdf
https://dsrl.eu/courses/pt/materials/PT2021-2022_Documentation_Template.doc

Basics

https://www.geeksforgeeks.org/
https://stackoverflow.com/
https://www.youtube.com/

For drawing

https://app.diagrams.net/

Other sites

https://www.javacodeexamples.com/
https://www.javatpoint.com/
https://howtodoinjava.com/
https://www.baeldung.com/parameterized-tests-junit-5
https://www.computerhope.com/

Math related

https://www.mathsisfun.com/
https://www.cuemath.com/
https://www.symbolab.com/
https://brilliant.org/wiki/polynomial-division/

Chinese website (don’t ask)

https://blog.csdn.net/shiniannian/article/details/104743742

Videos which helped in critical situations

https://www.youtube.com/watch?v=DYSPEkvTWig&t=343s
https://www.youtube.com/watch?v=EY973TC6AHQ
https://www.youtube.com/watch?v=BuW7y21FcYI&t=670s
https://www.youtube.com/watch?v=o4-ohj-Bm90