Technical University of Cluj-Napoca

Year 2016-2017

Department of Computer Science

Programming Techniques

1st Assignment

Student: Gyarmathy Tímea

Group: 30423

Table of Contents

[Specification of the problem 2](#_Toc477509048)

[The analysis of the problem 2](#_Toc477509049)

[Modeling 4](#_Toc477509050)

[Use cases 4](#_Toc477509051)

[Scenarios 4](#_Toc477509052)

[Design 5](#_Toc477509053)

[Design concepts 5](#_Toc477509054)

[Class diagram 6](#_Toc477509055)

[The Model 6](#_Toc477509056)

[The View 9](#_Toc477509057)

[The Controller 10](#_Toc477509058)

[The Main Class 11](#_Toc477509059)

[Graphical User Interface 11](#_Toc477509060)

[Implementation and testing 12](#_Toc477509061)

[Results 12](#_Toc477509062)

[Conclusions 13](#_Toc477509063)

[What have I learned? 13](#_Toc477509064)

[Further developments 13](#_Toc477509065)

[Bibliography 13](#_Toc477509066)

# Specification of the problem

Propose, design and implement a system for polynomial processing. Consider the polynomials of one variable and integer coefficients.

# The analysis of the problem

A polynomial is a mathematical expression consisting of a sum of terms. These terms, also defined as monomials, are each composed of a coefficient multiplied with one or more unknown variables, which are each raised to a certain power (if this power is 0, we get the constant term of the polynomial). These powers we define as the grade, the degree of the monomial. The general form of a single variable x and coefficients a0, . . ., an polynomial is the following:

anxn + an-1xn-1 + . . . + a2x2 + a1x + a0

We refer to n, the highest power of x (the largest degree of the constituting monomials) as the degree of the polynomial.

In our problem specification we will only work with polynomials following the next rules:

* The polynomials are of a single variable, denoted by x from this point on
* The powers of x are integers and also nonnegative numbers
* Coefficients are integer numbers as a convention, but as a bonus to the current implementation, real coefficients are also accepted and the polynomial operations work on these aswell

Polynomials are widely used in mathematics and science so a calculator for the basic operations that can be performed with them can prove to be a useful utility.

These basic operations will be implemented in this assignment:

Addition

To add two polynomials, we have to add the coefficients of the monomials with the same degree. The resulting polynomial will consist of these sums aswell as the single monomials from the operands.

E.g.: (x4 + 3 \* x2 + 20) + (7 \* x2 + 6 \* x) = x4 + 10 \* x2 + 6 \* x + 20

Subtraction

To subtract two polynomials, we have to subtract the coefficients of the monomials with the same degree, the second from the first. The resulting polynomial will consist of these terms aswell as the single monomials from the operands, where the monomials from the second one will be taken with the opposite sign.

E.g.: (x4 + 3 \* x2 + 20) **-** (7 \* x2 + 6 \* x) = x4 + (-4) \* x2 **-** 6 \* x + 20

Multiplication

To multiply two polynomials, we have to multiply every term from the first one with every term from the second one, and in the result the coefficients of the monomials with the same degree will be added. When we multiply two monomials, the result coefficient will be the product of the coefficients and the degree of the result will be the sum of degrees of these operands.

E.g.: (x4 + 3 \* x2 + 20) **\*** (7 \* x2 + 6) = 7 \* x4+2 + 6 \* x4 + 3\*7 \* x2+2 + 3\*6 \* x2 + 20\*7 \* x2 + 20\*6

= 7 \* x6 + 6 \* x4 + 21 \* x4 + 18 \* x2 + 140 \* x2 + 120

= 7 \* x6 + 27 \* x4 + 158 \* x2 + 120

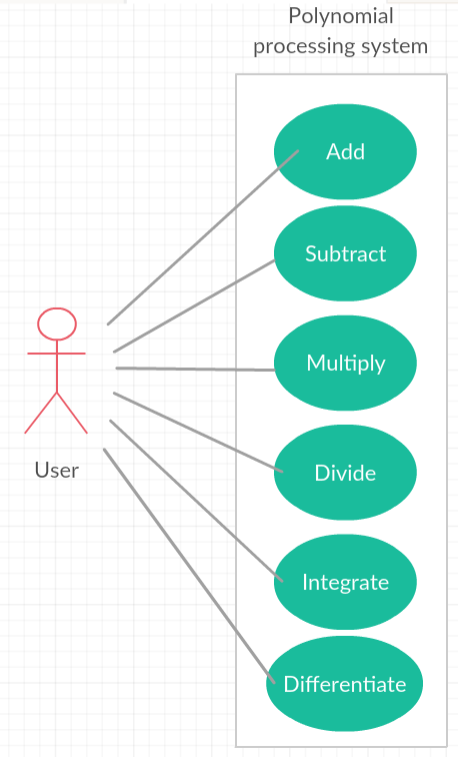
Division

To divide two polynomials, the algorithms is more complex. The basic principle instead, it resembles the division of integer numbers so we will have a result and a remainder at the end.

E.g.: (x4 + x3 + 4 \* x2 + 3 \* x + 20) **/** (x2 + x + 2) = x2 + 2 , remainder: x + 16

Integration

Integration is executed on a single operand, the first polynomial in the current implementation, and it follows the laws of function integration.

Differentiation

Differentiation is executed on a single operand, the first polynomial in the current implementation, and it follows the laws of function differentiation.

# Modeling

To be able to develop an application for the abstract concept of polynomials, we have to decompose this into elementary pieces, in our case, the monomials. This way, we have to first implement operations on monomials, as the smaller problem, and then construct a scheme on these to be able to perform operations with polynomials also.

# Use cases

The user (actor) can directly perform one of the six available operations on the input polynomials.

## Scenarios

Precondition: The user successfully launches the application.

Success:

1. The user introduces in the upper text field the coefficients of the first polynomial, in order, from a0 to an separated by only one whitespace
2. The user introduces in the lower text field the coefficients of the second polynomial, in order, from a0 to an separated by only one whitespace
3. The user chooses from the dropdown list the operation to be performed
4. If ready, clicks on the “Compute” button
5. The first input string is transformed internally into data modelled as polynomials, parsing and verifying whether the input was correct or not
6. The second input string is transformed internally into data modelled as polynomials, parsing and verifying whether the input was correct or not
7. The two polynomials are displayed in standard format to the user
8. Operations are performed on internal data
9. Result is returned in string format to the user interface and displayed

Alternative scenarios:

**First:** Bad data is input into text field 1. (it est, non-numeric characters, or more spaces between the coefficients), step 5 occurs, parsing is unsuccessful:

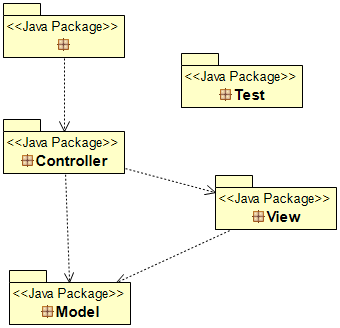
1. Error message displayed on a separate pop-up window that informs the user about the invalid input
2. User corrects mistakes
3. Return to step 4.

**Second:** Correct data is input into text field one but bad data is input into text field 2, step 5 occurs, parsing is unsuccessful for second polynomial:

1. Error message displayed on a separate pop-up window that informs the user about the invalid input, while the first polynomial is displayed in the standard format
2. User corrects mistakes
3. Return to step 4.

# Design

## Design concepts

The main concept of developing this polynomial processing system is based on Model – View – Controller design process which means firstly we will have three packages: Model, View and Controller, to which a Test package was added throughout the development process.

We will clearly have a Polynomial aswell as a Monom class, the latter having two subclasses related to the type of their coefficients: MonomInteger and MonomReal. These classes should be the Model to our whole system.

The View contains a single class, UI, which implements the user interface.

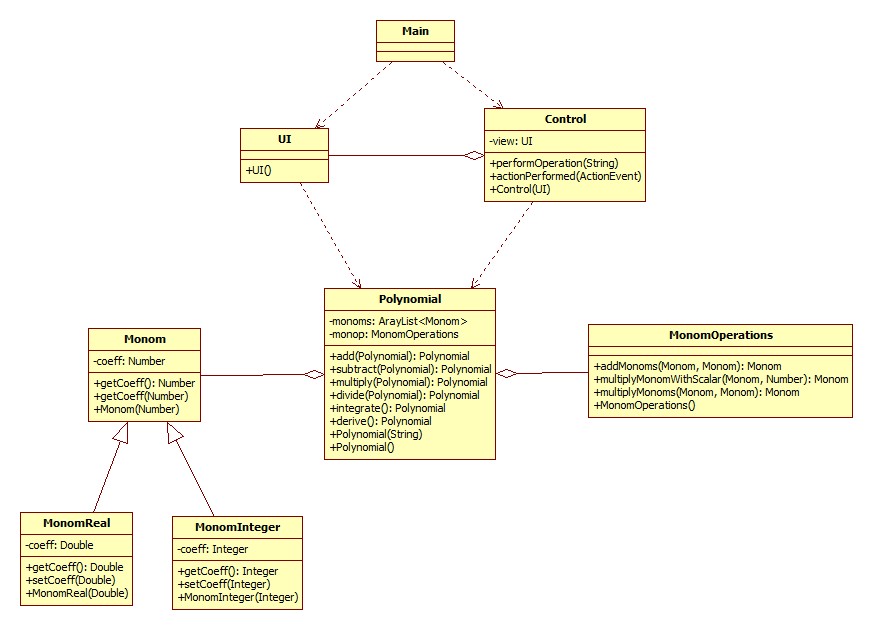
As we mentioned before, to ease our job, we will decompose the problem into smaller ones, thus implementing operations on monomials in the MonomOperations class, which belongs to the Controller package. This package has one more class, Control, which directly connects the view with the model, thus controlling our whole system.

The Test package, as the name suggests, should contain the tests which were executed throughout the development process, in our case, PolynomialTest class is a Junit test case, and verifies the correctness of the polynomial operation algorithms.

Near all these we also have a separate Main class which contains only the main function that runs the application.

## Class diagram

The summarized class diagram looks like the following:

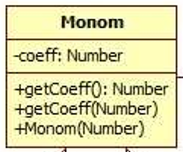


Some methods are not represented here because they are not relevant for the modeling, they are only helping the implementation (like getters and setters of the toString method).

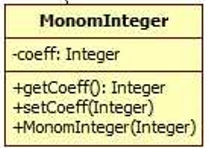
A complete class diagram is added to the end of this chapter.

### The Model

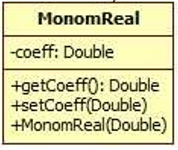
#### Class Monom

This class is the basic unit of this project. It represents a monomial, a term, a node in a polynomial. It has a private coeff member, an object of type Number. It contains the getter and setter for it, and also a constructor which takes a Number as a parameter and sets it the current. This class has no parameterless constructor, because it would be meaningless to create a monomial without a coefficient.

#### Class MonomInteger

It extends class Monom but it has coeff of type Integer. Its constructor calls the superclass constructor but has as a parameter an Integer, and sets the coeff accordingly. Also contains setter and getter for it, the latter overriding the superclass getter.

#### Class MonomReal

Similarly to class MonomInteger, it is based on coefficients of type Double.

#### Class Polynomial

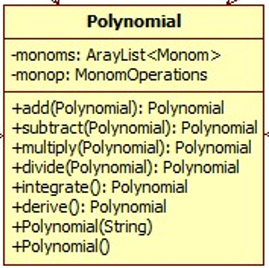
The most complex class of our project. It models a polynomial, having as a component an ArrayList of monomials:

**private** List<Monom> monoms = **new** ArrayList<Monom>();

ArrayList implements the interface List, and consists of ordered nodes.

Then the operations on monoms are included as a component:

**private** MonomOperations monop = **new** MonomOperations();

All the available operations on polynomials are implemented within this class, in an object oriented manner, all the methods taking the current polynomial as one operand, and getting the second one as a parameter.

It has an empty constructor, but the main constructor of this class, which takes a string as an argument, resolves the number parsing and creates from a string an array of accordingly created Monom-s (MonomInteger or MonomDouble).

##### The idea of implementation of the operational methods

Let’s take first the add method, and follow its implementation step by step.

1. **public** Polynomial add(Polynomial polyIn)

{

ArrayList<Monom> monomsList1 = monoms;

ArrayList<Monom> monomsList2 = polyIn.getPoli();

So far, as we enter the method we identify our two operands and obtain their monomial lists. As we can see, to monomsList1 is assigned directly monoms, which is the array list of monomials from the current object. monomsList2 gets the ArrayList of the parameter Polynomial polyIn. getPoli() is the getter method for the polynomial’s ArrayList and returns it directly in this form.

1. In the next step we declare an array of Monom objects in which we will construct our result polynomial.

// Make resulting array big enough to fit both polynomials

Monom[] resultMonoms = **new** Monom[Math.*max*(monomsList1.size(),monomsList2.size())];

1. From here, we perform the addition, which means we forst add to resultMonoms the first list of monoms, stored in monomsList1:

// Copy the first monom array into results

**for**(Monom monom1: monomsList1)

{

**int** grade = monomsList1.indexOf(monom1);

resultMonoms[grade]= monom1;

}

1. Then we add the second list correspondingly. We iterate through monomsList2, and take the monoms one by one. If a monom of that grade does not yet exist in our result array, then we add it, if it does, then we add them with the operation addMonoms. This method is implemented din the class MonomOperations, and it adds the coefficients of the two operands, and returns a node of the corresponding type.

// Add the second monom array to results

**for**(Monom monom2: monomsList2)

{

**int** grade = monomsList2.indexOf(monom2);

// If there's NO monom of that grade, just put the object as is

**if**(resultMonoms[grade] == **null**)

{

resultMonoms[grade] = monom2;

}

// Add values of the two monoms if one exists from the first list

**else**

{

Monom monom1 = resultMonoms[grade];

resultMonoms[grade] = monop.addMonoms(monom1, monom2);

}

}

1. To return the result, first we convert the array of Monom-s into an ArrayList of Monom-s, then construct a Polynomial based on it, and then return the resulting Polynomial object.

// return the result

ArrayList<Monom> resultsAL = **new** ArrayList<Monom>(Arrays.*asList*(resultMonoms));

Polynomial poliRes = **new** Polynomial();

poliRes.setPoli(resultsAL);

**return** poliRes;

}

The idea of polynomial subtraction is the same, but because subtracting is reused at the integration of a polynomial, steps (2) – (4) have been implemented in a separate method:

**private** ArrayList<Monom> subArrayMonoms(ArrayList<Monom> monomsList1,

ArrayList<Monom> monomsList2)

And it is only called in the subtract method for polynomials upon the declared resultArray of type ArrayList from which the resulting polynomial will be constructed.

The method of multiplication of polynomials is implemented based exactly on the principle of it: we iterate through the first polyinomial’s monoms and at every term we iterate through the second polynomial aswell. We multiply the terms by calling the methop multiplyMonoms from class MonomOperations and construct a result array such that terms with the same degree are added (with the help of the method addMonoms). The result array is then converted into a Polynomial as seen in the case of addition.

The operation division of polynomials was the hardest one to implement from my point of view. It follows the algorithm for long polynomial division[[1]](#footnote-1) and uses four helper methods, declared as private in this, Polynomial class:

**private** **int** degree(ArrayList<Monom> monomList)

**private** ArrayList<Monom> shiftRight(ArrayList<Monom> monomList, **int** shift)

**private** ArrayList<Monom> multiplyWithMonom(ArrayList<Monom> monomList, Monom monom)

And the method subArray(Monom, Monom) already mentioned before.

It returns an array of two Polynomials, the element on index [0] being the result of division, the quotient, and the element on index [1] the remainder.

The functions for deriving and integrating one polynomial are very easily implemented, as helper were reused in them. For example the integration of the current Polynomial:

**public** Polynomial integrate(){

ArrayList<Monom> monomsList = (ArrayList<Monom>) monoms;

Monom[] resultMonoms = **new** Monom[monomsList.size() + 1];

resultMonoms[0] = **new** Monom(0);

// Integrate all monoms one by one

**for**(Monom monom: monomsList)

{

**int** grade = monomsList.indexOf(monom);

resultMonoms[grade + 1]= monop.multiplyMonomWithScalar(monom,1.0/(grade+1));

}

// return the result

ArrayList<Monom> resultsAL = **new** ArrayList<Monom>(Arrays.*asList*(resultMonoms));

Polynomial poliRes = **new** Polynomial();

poliRes.setPoli(resultsAL);

**return** poliRes;

}

### The View

#### Class UI

It contains the whole implementation of the graphical user interface, it extends Jframe and it has a list of components from which the constructor ‘constructs’ the user interface.

It has a minimalist style, consisting of only one frame, so when the user opens the application, all the elements appear at once.

It contains a JLabel with the instructions for the user to use the application. Below this, the input box for the first and second polynomial is shown, as well as the output boxes. After the user enters the polynomial according to the instructions, and presses the button, not only the result is displayed but also the input polynomials in conventional style returned by the toString method from the class Polynomial. This toString is written explicitly and overrides the method from Object superclass.

There appears a combo box with the list of operations that can be performed, and below this, the “Compute” button, for which an action listener is also added in order to be able to respond to the user.

**public** **void** addComputeListener(ActionListener comp) {

compute.addActionListener(comp);

}

The other methods present in this class are all getters and setters for the dynamic graphical elements.

### The Controller

#### Class MonomOperations

This class has been mentioned several times already, so its function is clear. It contains methods for basic operations directly on and between monomials:

* Add two monomials
* Multiply monomial with scalar
* Multiply two monoms

#### Class Control

This class represents the link between the graphical user interface and the model. It implements ActionListener and has as component the UI. By its constructor, it listens to the “Compute” button from the UI. If an event occurred, i. e. the button was pressed, it redirects to an operation based on the selection from the combo box also from the UI.

### The Main Class

It contains only the method main, indispensable to be able to run the application:

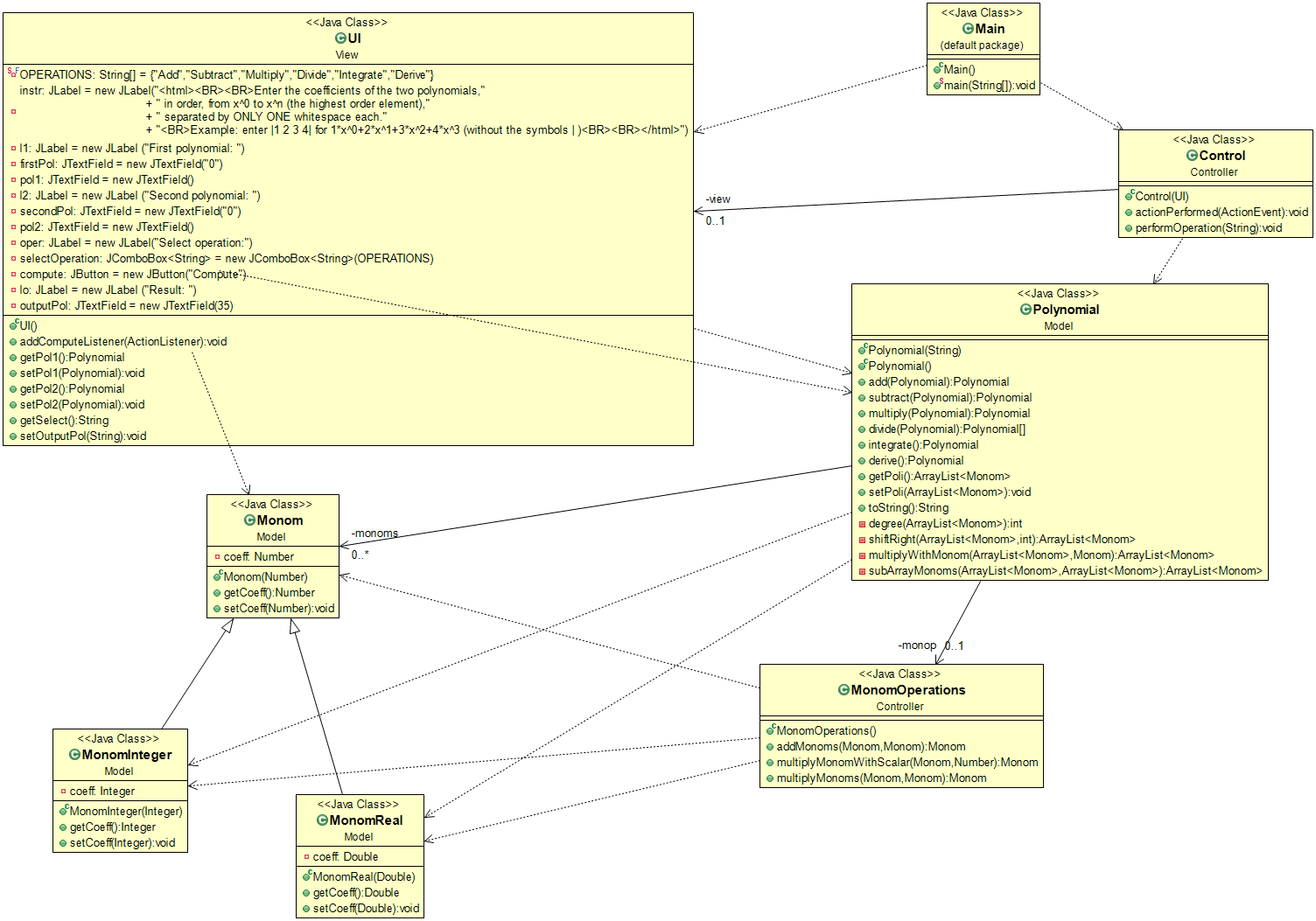
**public** **static** **void** main(String[] args) {

//calling the application user interface and control

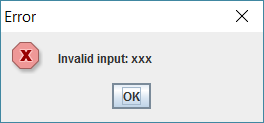
UI ui = **new** UI();

Control ctrl = **new** Control(ui);

ui.setVisible(**true**);

}

## Graphical User Interface

As already shown, it has a minimalist style, and the user has to follow the intstructions precisely to input the polynomials correctly. In case of bad input a pop-up window with an error message and the bad data is shown and the user has to return and correct his mistakes.

The problem specification states that the polynomials should have integer coefficients, but this application opens up to real coefficients also, thus giving more possibilities for the user to make use of it, and perform operations on a wider variety of polynomials.

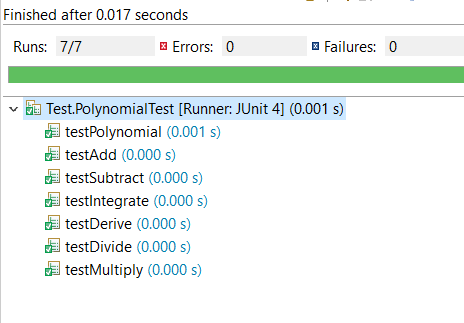
Even if the input from the user is gotten in an unconventional style, i.e. the user has to input only the coefficients for x0, x1, . . ., xn in order, separated by only one whitespace, when the button is pressed, the result and the two inputs aswell are displayed in the standard format.

Example input: 1 2 3 4

Meaning: 1 \* x0 + 2 \* x1 + 3 \* x2 + 4 \* x3 = 4x3 + 3x2 + 2x + 1

Standard form: 4\*x^3+3\*x^2+2\*x^1+1\*x^0

# Implementation and testing

Implementation started from modeling the problem and drawing the class diagrams, from where I first implemented the structure of the Model package. When it was done, I wrote the constructor for the Polynomial class, and created a JUnit to test if it constructs correctly the polynomial from a String.

As the operational methods were implemented over time, they were gradually added to this JUnit Test file, called PolynomialTest which I added to a separate package Test.

This class has seven test methods, six for testing the operations between and on polynomials, and one for testing the constructor.

These tests rely on the principle of “golden data”. This means, firstly that data is constructed from the method which needs to be tested, and then a “golden” one is made, which should always be the correct result. Then these two are then compared and if they are not equal, the test fails.

For example the test method for addition looks like this:

@Test

**public** **void** testAdd() {

Polynomial firstPol = **new** Polynomial("1 2 3 4");

Polynomial secondPol = **new** Polynomial("3 2 1");

//our polynomial is created

Polynomial resultPol = firstPol.add(secondPol);

//golden polynomial is created, the correct result for addition

Polynomial goldenPol = **new** Polynomial("4 4 4 4");

//comparison

**for**(**int** i=0; i<goldenPol.getPoli().size();i++){

Monom m1 = goldenPol.getPoli().get(i);

Monom m2 = resultPol.getPoli().get(i);

**if** (!m1.getCoeff().equals(m2.getCoeff())){

*fail*("Elements are different");

}

}

}

# Results

Even if the problem in question may have been strange to be implemented in Java programming language, it resulted in a basic, easy to use application for polynomial processing. The input mode is fast, altough it puts some constraints on the data that can be input for now, for example polynomials with very large degree are really hard to input this way, this can be solved in further developments. But for now, the application is perfectly suitable for small and fast calculations.

# Conclusions

## What have I learned?

I think I have struggled enough with this assignment to understand that anything can be implemented in object oriented approach if modelled correctly. My code may not be the most performant, but this being a pretty large project, I learned to organize it, to comment it and to use the right indentation. I have also seen the benefits of testing the code throughout the process of development, because problems were discovered at the right time.

## Further developments

My application could be reorganised in such a way that the input is also in standard format, which of course takes more time to be input but it creates the possibility of inputting monomials with very large degree. Beside this, maybe in the future, more operations could be added, like solving the polynomials and finding their roots. This application already works with double coefficients so maybe this feature will come in handy.

# Bibliography

1. For programming issues: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>
2. Long polynomial division: <https://rosettacode.org/wiki/Polynomial_long_division>
3. Formal description of polynomials: <https://en.wikipedia.org/wiki/Polynomial>
4. Class diagram description: <https://en.wikipedia.org/wiki/Class_diagram>
5. Last semester’s OOP code on how to use Swing to create user interface and how to establish the connections in an MVC model

1. Pseudocode taken from: https://rosettacode.org/wiki/Polynomial\_long\_division [↑](#footnote-ref-1)