

March 2016



Laboratory Assignment 1

Polynomial processing

Farcas Silviu Vlad

30425/1

Contents

1. Introduction

1.1. Task Objectives	4
1.2. Personal Approach	4

2. Problem Description

2.1. Problem Analysis	5
2.2. Modeling	5
2.3. Scenarios	6
2.4. Use cases	7

3. Design

3.1. UML Diagrams	9
3.2. Data Structures	13
3.3. Class Design	14
3.4. Interfaces	20
3.5. Relations.....	20
3.6. Packages	21
3.7. Algorithms	21
3.8. User Interface	25

4. Implementation and Testing26

5. Results26

6. Conclusions

6.1. What I've Learned	26
6.2. Future Developments.....	27
 7. Bibliography	 27

1. Introduction

1.1 Task Objective

The task of this assignment is defined as follows:

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

Thus, we should include in our system basic polynomial processing functions, like addition, subtraction, multiplication, division, differentiation, integration, root-finding and plotting the graph.

1.2. Personal Approach

The aim of this paper is to present one way of implementing the required system, based on a simple and minimal GUI that provides the user with the possibilities of computing all the above mentioned functions on a given set of two polynomials. Note that polynomials will be inputted as an array of integer numbers. For example, for

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

$$a_0 \neq 0, \quad n \in \mathbb{N}$$

the user will input $a_n \ a_{n-1} \ \dots \ a_1 \ a_0$.

2. Problem Description

2.1. Problem Analysis

First of all, what is a polynomial?

Wikipedia:

In mathematics, a polynomial is an expression consisting of variables and coefficients which only employs the operations of addition, subtraction, multiplication, and non-negative integer exponents. An example of a polynomial of a single variable x is $x^2 - 4x + 7$. An example in three variables is $x^3 + 2xyz^2 - yz + 1$.

A polynomial in a single indeterminate x can always be written (or rewritten) in the form

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

$$a_0 \neq 0, \quad n \in \mathbb{N}$$

A polynomial one variable is strongly associated with its coefficients. This will be strongly reflected in our final system, as I will model each polynomial as ultimately be described by only two items: its coefficients and its degree. We should be able to approach addition, subtraction and all other operations on polynomials in this manner: working with two arrays of coefficients, we should create another array of coefficients which will be associated with a new polynomial.

2.2 Modeling

A polynomial can be modeled by specifying its list of coefficients, from where we infer its degree. Based on the observations above, we can already envision the following model: some resources allocated for

storing polynomials (including arrays for storing coefficients and integers for degree), some other resources build for handling functions (addition, subtraction, multiplication, division, differentiation, integration (plotting functions will be the role of the GUI in strong association with the function resources)), some others for the Graphical User Interface (GUI) and some others for the Input/Output business logic. Ultimately, these resources will translate into packages and classes. We can spot already a models package, a controller package and a views package. However, we can also spot a “little grand” detail. Although the problem specification implies integer coefficient polynomials, if we want to implement division we will need to handle polynomial with real coefficients, as presented in the example below:

$$\frac{x^3 + 3}{2x^2 + 1} = 0.5x \text{ r: } -0.5x + 3$$

The same goes for integration:

$$\int x dx = 0.5x^2 + C$$

Note that we will not output the constant (it will be truncated).

These “little grand” notes will have a significant impact on our project, since we will have to design polynomials with integer and real coefficient, too. We will use a general class Coefficient which will be extended by a CoefficientInt and a CoefficientReal class, but more on that later.

2.3 Scenarios

Say a given user inputs a polynomial. It wants to perform addition, so it presses the button “+” (add). Immediately, the user should see the mathematical form of the polynomial resulted by addition of the two inputted polynomials. One can notice that the polynomial class will need

an overwriting of the `toString()` method so that it prints a polynomial represented by a series of coefficients in the mathematical form. The same goes for subtraction. When performing division, a user should see both the quotient and the rest, with a mention that the result will have real coefficients. When performing integration, we take into account the indefinite integral, which produces a constant at the end of the result. The constant will NOT be taken into account (it will not be displayed on screen).

Now say that the user wants to find the root around the point x . If the polynomial has such root, it should display it; if not, it should display a proper error message (root not found). If we want to plot the graph, a new window will pop up with the function being represented on a $(-5, 5) \times (10, 10)$ real domain. Note that the domain is not designed to be resizable, i.e., user selected. The domain $(-5, 5) \times (10, 10)$ should be enough to display any polynomial of degree at most 10 with coefficients less than 20. Any portion of the function outside this interval will not be visible.

If the user wants to perform an operation, but no polynomials are provided, an exception will be generated which is not handled by the system. From this point of view, there is a little side of error proneness to our program. Also, if the user enters non-numeric characters in the polynomial field, an exception will be generated which will not be handled. Also, if the user wants to evaluate a function at a point x , or find a root around x , but no x is inputted, the same as above happens. This is a good starting point for future developments, i.e., making the program take into account the “stupidity” of the user.

2.4 Use cases

There is only one actor in our system: the polynomial user. This user can do one of the following operations:

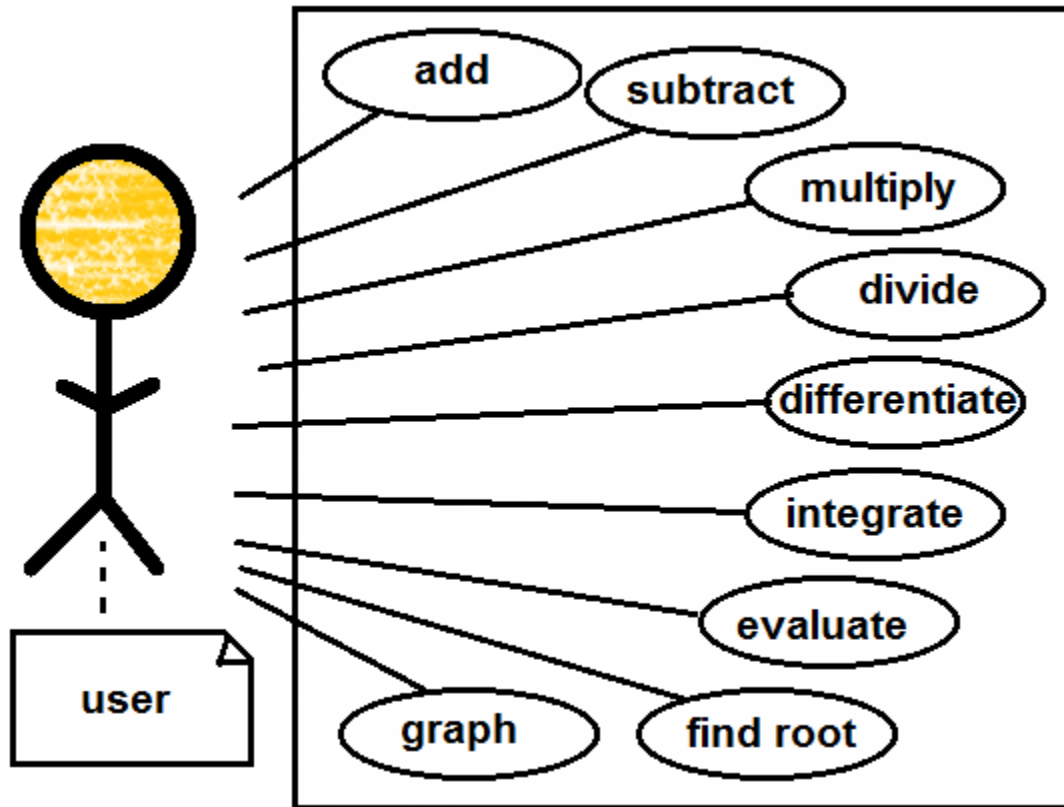
- add. This function is defined on two polynomials and returns a third. Used with integer coefficient polynomials.
- subtract. This function is defined on two polynomials and returns a third. Used with integer coefficient polynomials (although a version of this function used for division is designed for real coefficient polynomials).
- multiply. This function is defined on two polynomials and return a third. Used with integer coefficient polynomials (although a version of this function used for division is designed for real coefficient polynomials).
- divide. This function is defined on two polynomials and return a third. Used with real coefficient polynomials
- differentiate. This function is defined on two polynomials and return a third. Used with integer coefficient polynomials.
- integrate. This function is defined on two polynomials and return a third. Used with real coefficient polynomials.
- evaluate at x. Evaluates polynomial at x.
- root around x. Find a root of polynomial around x using the bisection method (again, this is the subject of future development since we could use other algorithms for root-finding; bisection method is relatively slow in converging).
- graph. This function plots the selected polynomial on the $(-5, 5) \times (10, 10)$ domain.

3. Design

3.1. UML Diagrams

In the following we will present Use Case diagram, Class Diagram and Sequence Diagram.

3.1.1. Use Case Diagram



Use Case Description

The user enters two polynomials and a point x . The user selects the operation to perform. The system computes and displays the result. The user is happy.

Trigger

The user presses one of the buttons.

Actors

The user

Preconditions

The user enters polynomials as an array of integer numbers, and point x as a real number.

Goals (Successful Conclusion)

Provide the user with computed data.

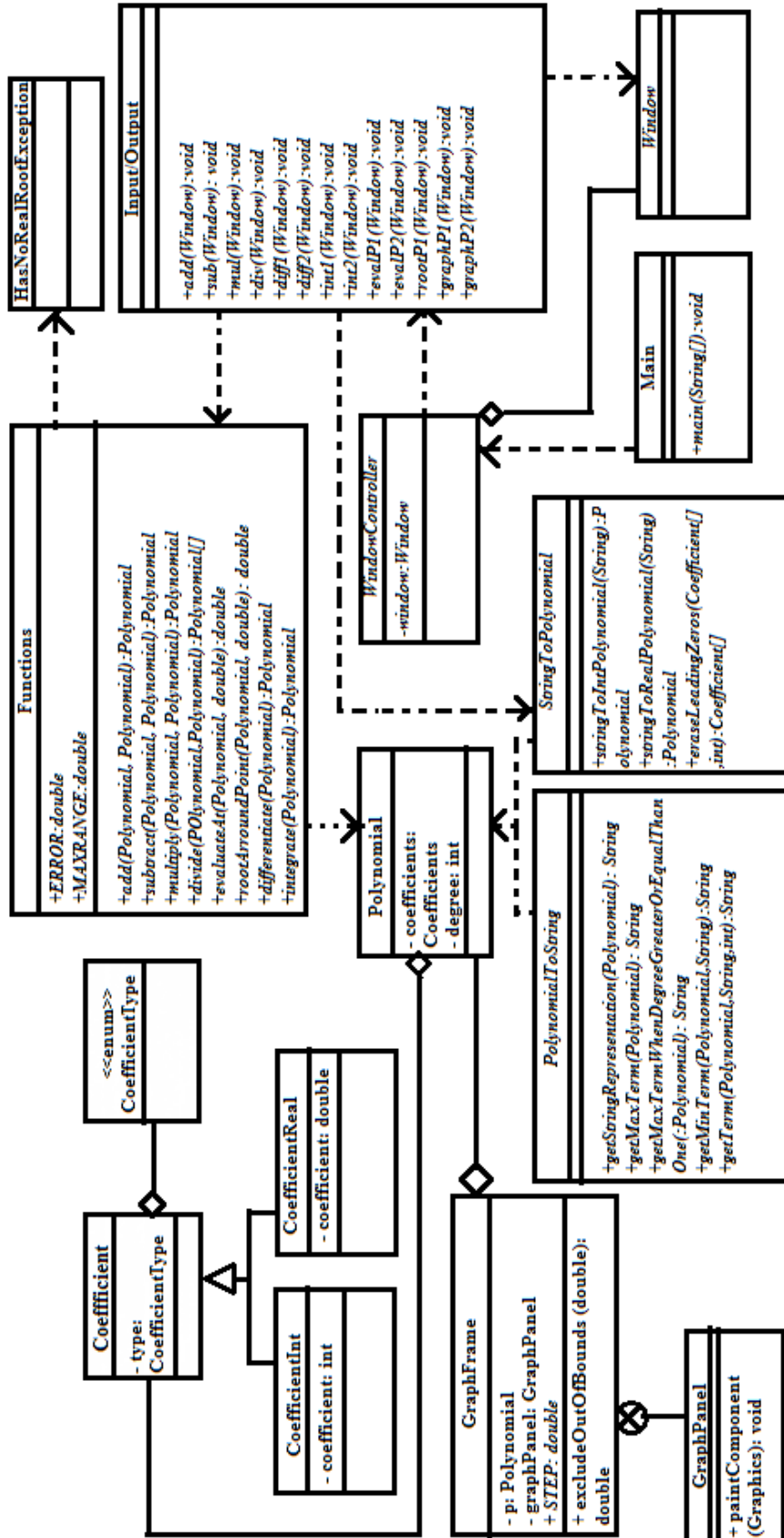
Failed Conclusion

1. Non-numerical string of coefficients provided.
2. Non-numerical x provided.
3. Not provided sufficient input to perform specific operation

Steps of execution

1. The user enters data.
2. The user presses the operation button.
3. Operation is performed
4. Result is displayed

3.1.2. Class Diagram (next page)

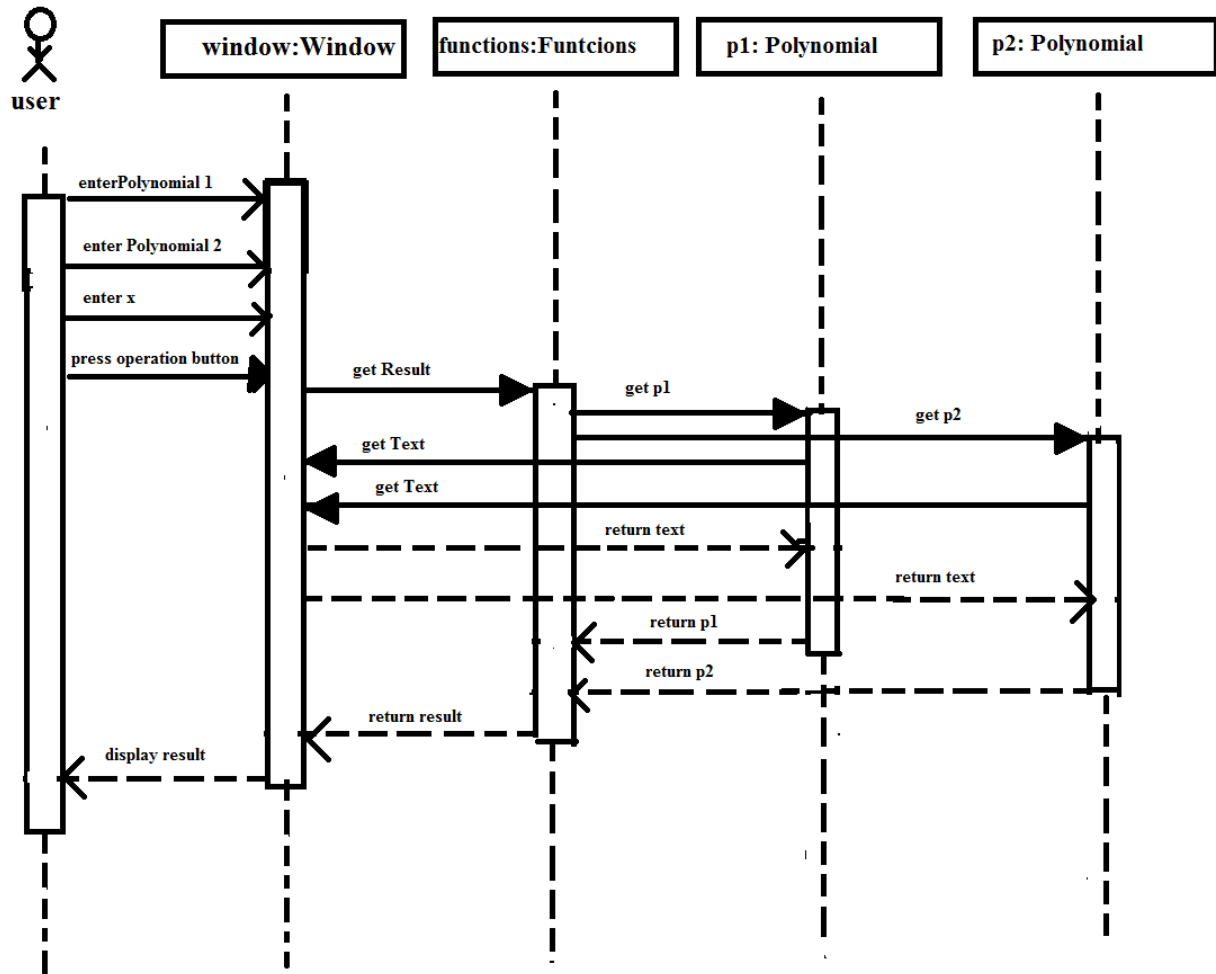


The class diagram shows the modelling of our problem into classes. There are 13 classes and one enumeration. One must know that not all fields and methods were represented on the diagram, only the ones that are important to our implementation (for example, getters/setters are not shown). Also, not all dependencies have been drawn, otherwise it would make the diagram unreadable. I have used inheritance for class `CoefficientInt` and `CoefficientType`, which inherit class `Coefficient`. I have also used aggregation between:

- 1) `Coefficient` and `CoefficientType` enum, since `Coefficient` has a field of type `CoefficientType`.
- 2) `polynomials` and `Coefficients`, since `polynomials` contain an array of coefficients
- 3) `GraphFrame` and `Polynomials`, since `GraphFrame` contains a polynomial
- 4) `WindowController` and `Window`, since `WindowCotroller` holds a `Window` object

The rest of the relations are dependencies.

3.1.3. Sequence Diagram (next page)



The sequence diagram presents the processing of polynomials in a sequential manner. First, the user presses a button, then the polynomials are read, the operation is performed and then the result is displayed.

3.2. Data Structures

Our project mainly utilizes primitive data types, like int and double. Also, it makes use of an enumeration used to describe the type of coefficients: CoefficientType can be INT or REAL. Polynomials are linked to an array of coefficients and a degree. I have decided to use mere arrays instead of other data structures (like ArrayList<>, for example) because retrieving and storing are performed with less

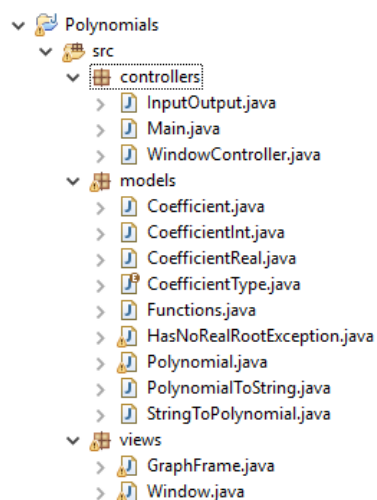
commands, which would otherwise increase unnecessarily the dimension of our code and also because our list of coefficients is not variable in dimension (once a polynomial is created, its coefficients remain the same, no coefficients are added/deleted). However, using data structures would significantly improve the object orientation of our system, but this is topic for future development.

3.3. Class Design

The problem domain was decomposed into sub-problems, and for each sub-problem a solution was build. By assembling the solutions we obtain the general solution to our problem. The sub-problems were:

- how to correctly model a polynomial and correctly model functions on polynomials and get the desired result?
- how will be implemented the user interface?
- how can we assure user interaction with our models?

For each of the above mentioned, we came up with a package that would support the features of polynomial processing, namely the mathematical model of polynomials and functions, the user interface and user interaction with our model. These packages follow the Model-View-Controller software architecture and they are as follows:



We first started designing our **model** package. The coefficients our polynomials use are integer and real. We will have a **CoefficientType** enumeration {REAL, INT} designed to handle this situation. Then, we will make a general **Coefficient** class with one attribute:

```
private CoefficientType type;
```

When constructing a **Coefficient**, we will parse its type:

```
public Coefficient(CoefficientType type) {
    this.type = type;
}
```

Then, the **Coefficient** class will be extended into a **CoefficientInt** class and a **CoefficientReal** class. These classes will inherit the type from **Coefficient**, plus they will have a new attribute, the coefficient itself, be it integer or real.

```
public class CoefficientInt extends Coefficient {
    private int coefficient;

    public CoefficientInt(int coefficient) {
        super(CoefficientType.INT);
        this.coefficient = coefficient;
    }
}
```

The **Polynomial** class will be constructed from an array of **Coefficients**, and will have as attributes the degree and the array of coefficients.

```
public class Polynomial {
    private int degree;
    private Coefficient[] coefficients;

    public Polynomial(Coefficient[] coefficients) {
        this.degree = coefficients.length - 1;
        this.coefficients = coefficients;
    }
}
```

This class also overwrites the `toString()` method and calls a static method from class **PolynomialToString**, which will be detailed in what follows.

```
public String toString() {
    return PolynomialToString.getStringRepresentation(this);
}
```

The **PolynomialToString** class is responsible with displaying a given Polynomial object as mathematical sequence of terms. It is not a light-weighted class, since displaying polynomials in mathematical form is quite tricky (keeping track of the sign of the term, displaying the last term, displaying the first term etc.). All the function in this class are static, since this is a helper class and there is no need to instantiate one object to make use of its capabilities. The toString() method in the Polynomial class calls the getStringRepresentation() method:

```
public static String getStringRepresentation(Polynomial p) {
    String poly = "";
    int i;
    for (i = 0; i <= p.getDegree(); i++) {
        if (i == 0) {
            poly = getMaxTerm(p);
        } else if (i == p.getDegree()) {
            poly = getMinTerm(p, poly);
        } else {
            poly = getTerm(p, poly, i);
        }
    }
    return poly;
}
```

Each of the methods used above are detailed in the PolynomialToString class. They deal with the particular cases of mathematical representation of polynomials.

Because we output the data using this class, we will also need to perform the backward operation (getting a Polynomial object from a string of coefficients) using another class for inputting the data. This class is called **StringToPolynomial**. Again, all methods in this class are static.

If we want to parse a string for constructing a polynomial with integer coefficients, we use the StringToIntPolynomial method:

```
public static Polynomial stringToIntPolynomial(String line) {
    String[] coefficientsAsString = line.split(" ");
    int i, degree = coefficientsAsString.length - 1;
    Coefficient[] coefficients = new Coefficient[degree + 1];
    for (i = 0; i < coefficientsAsString.length; i++) {
        CoefficientInt c = new
CoefficientInt(Integer.parseInt(coefficientsAsString[i]));
        coefficients[i] = c;
    }
}
```



```

        return new Polynomial(eraseLeadingZeros(coefficients, degree));
    }

```

If we want to parse a string for constructing a polynomial with real coefficients, we use the `StringToRealPolynomial` function:

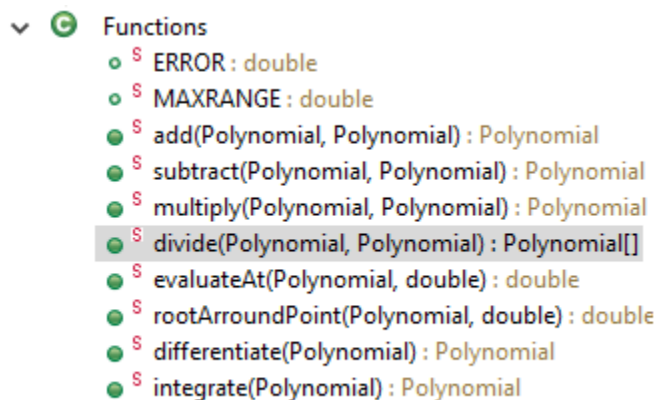
```

public static Polynomial stringToRealPolynomial(String line) {
    String[] coefficientsAsString = line.split(" ");
    int i, degree = coefficientsAsString.length - 1;
    Coefficient[] coefficients = new Coefficient[degree + 1];
    for (i = 0; i < coefficientsAsString.length; i++) {
        CoefficientReal c = new
CoefficientReal(Double.parseDouble(coefficientsAsString[i]));
        coefficients[i] = c;
    }
    return new Polynomial(eraseLeadingZeros(coefficients, degree));
}

```

There is also a method for erasing leading zeros in a string of coefficients, in case the user enters wrong data. This function is also used in later development of the `Functions` class, as we will see below.

The **Functions** class is probably the most complex class of our design. It models all the possible functions on polynomials add, subtract, multiply, divide, differentiate, integrate, find root, evaluate and plot:



```

Functions
  ERROR: double
  MAXRANGE: double
  add(Polynomial, Polynomial): Polynomial
  subtract(Polynomial, Polynomial): Polynomial
  multiply(Polynomial, Polynomial): Polynomial
  divide(Polynomial, Polynomial): Polynomial[]
  evaluateAt(Polynomial, double): double
  rootArroundPoint(Polynomial, double): double
  differentiate(Polynomial): Polynomial
  integrate(Polynomial): Polynomial

```

The algorithms involved will be discussed later, but for now, it is useful to know that the methods `add`, `evaluateAt`, `rootArround` and `differentiate` take as input parameters only polynomials with integer coefficients; the methods `divide` and `integrate` take as parameters only polynomials with real coefficients and the methods `multiply` and `subtract` take as

parameters polynomials with both integer and real coefficients since they are used in the divide implementation.

One of the methods in the Functions class can throw a **HasNoRealRootException**, in case root finding is impossible, but more on that later.

Then, passing to the **views** package, one can spot the **Window** class. This class is responsible for the main panel of our app, where the user enters data and presses buttons. There are a lot of fields for this class, each with a specific function (text-fields, panels, buttons). The main windows uses a BorderLayout (X-axis oriented) for displaying two panels: the left panel (used for input data and output results) and the right panel (used for button menu). The Left panel uses a 4x2 GridLayout and the right panel uses a 4x4 GridLayout. Also, there are action listeners setters for all the buttons in the menu.

The **GraphFrame** class launches a new plotting window when the user presses the graph button. Its fields are:

```
private Polynomial p;
    private GraphPanel graphPanel;
    public static double STEP = 0.01;
```

The inner **GraphPanel** class extends JPanel and draws using paintComponent(Graphics g) method the graph of the Polynomial p. The algorithm of the graphing will be detailed later. For now, it is useful to know that it makes use of the method excludeOutOfBounds (detailed below) in order to normalize polynomial values outside the domain.

```
public double excludeOutOfBounds(double x) {
    while (((Functions.evaluateAt(p, x) * 20 < -200) ||
(Functions.evaluateAt(p, x) * 20 > 200) && (x <= 5))) {
        x += STEP;
    }
    return x;
}
```

Moving to the **controller** package, we spot the class **InputOutput**. This is the class that is responsible for the logic of processing input data, computing the result and displaying the result. It does this in dependency

with the classes `StringToPolynomial`, `Window` and `Functions`. It has 14 methods:

```

InputOutput
  S add(Window) : void
  S sub(Window) : void
  S mul(Window) : void
  S div(Window) : void
  S diff1(Window) : void
  S diff2(Window) : void
  S int1(Window) : void
  S int2(Window) : void
  S evalP1(Window) : void
  S evalP2(Window) : void
  S rootP1(Window) : void
  S rootP2(Window) : void
  S graphP1(Window) : void
  S graphP2(Window) : void

```

We will give as example the `add(Window)` method and the `graphP1(Window)` method:

```

public static void add(Window window) {
    window.getOutput()
        .setText(
            Functions

            .add(StringToPolynomial.stringToIntPolynomial(window.getInput1().getText()),
                StringToPolynomial.stringToIntPolynomial(window.getInput2().getText()))
                .toString());
}

public static void graphP1(Window window) {
    new
    GraphFrame(StringToPolynomial.stringToIntPolynomial(window.getInput1().getText()), "GraphP1");
}

```

The **WindowController** class resolves the link between `Window` buttons and `InputOutput` methods by assigning to each `Window` button a specific listener which will call an `InputOutput` function when triggered. It has one field: a `Window` object that is initialized in the constructor. For example, for the button `add`, we have:

```

window.setAddButtonListener(new AddButtonListener());

```

provided the inner class:

```
public class AddButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        InputOutput.add(window);
    }
}
```

The **Main** class is completely reproduced below:

```
public class Main {
    public static void main(String[] args) {
        new WindowController(new Window("Polynomials"));
    }
}
```

This concludes a quick introduction into the aspect of our class design.

3.4. Interfaces

Our design does not make use of Interfaces.

3.5. Relations

In Object Oriented Programming, it is a good habit to design classes as loosely coupled as possible, in order to avoid error propagation. I tried to stick with this principle and, with few exceptions, I think I managed to fulfill this requirement. In this design, one can find the following relations: inheritance, aggregation, dependency and innerness. We will analyze each of them below.

- inheritance: CoefficientInt, CoefficientReal extend from Coefficient
- aggregation: Coefficient has a COefficientType, Polynomial has an array of Coefficient, GraphFrame has a Polynomial, WindowController has Window
- dependency: Functions is dependent on Polynomial, Functions is dependent on HasNoRealRootException, PolynomialToString and StringToPolynomial are dependent on Polynomial, Main is dependent on WindowController, InputOutput is dependent on Functions, StringToPolynomial and Window.

- innerness: GraphFrame contains GraphPanel, WindowController contains AddButtonListener, SubButtonListener, MulButtonListener, DivButtonListener, Diff1ButtonListener, Diff2ButtonListener, Int1ButtonListener, Int2ButtonListener, EvalP1ButtonListener, EvalP2ButtonListener, RootP1ButtonListener, RootP2ButtonListener, GraphP1ButtonListener, GraphP2ButtonListener.

3.6. Packages

The design follows the Model-View-Controller (MVC) architecture. It has three packages: models, views, controllers.

3.7. Algorithms

We are mainly interested in algorithms used for operation processing and graphical representation. First, we start with the operations used for functions performed on polynomials.

The **add** method has a rather straightforward approach. It adds the polynomials, term by term, and returns the resulting polynomial. In this process, it sets the degree of the result to the highest degree of the polynomials.

The **subtract** method is a variation of the add method, but instead of adding term by term, it subtract term by term.

It is worth a while to take a look at the **multiply** algorithm (for example, for integer coefficients polynomials):

```
for (i = 0; i <= degree; i++) {
    coefficients[i] = new CoefficientInt(0);
    for (j = i; j >= 0; j--) {
        if (j <= p1.getDegree() && (i - j) <= p2.getDegree()) {
            ((CoefficientInt) coefficients[i])
                .setCoefficient(((CoefficientInt) coefficients[i]).getCoefficient()
                    + ((CoefficientInt) p1.getCoefficients()[j]).getCoefficient()
                    * ((CoefficientInt) p2.getCoefficients()[i - j]).getCoefficient());
        }
    }
}
```

```

    }
}

```

Take a simple example:

$$(x^3 + x) * (x + 1) = (x^4 + x^3 + x^2 + 1)$$

What were the steps in performing our operation?

The output is the result of combinations of sums of terms. The first term (index 0) is obtained by multiplying the term (0) (x^3) from p1 with the term (0) (x) from p2 and we get x^4 . However, second term (index 1) is obtained by adding the multiplication of the term (0) (x^3) from p1 with the term (1) (x) from p2 with the multiplication of the term (1) from p1 (0, it is not visible) with the term (0) from p2 (x). We get $x^3 + 0 = x^3$. And so on and so forth for other terms of the result. This example better illustrates the two for loops presented in the code.

The **division** algorithm also proceeds from the real mathematical model of the operation of division. Here is a short snippet of the code:

```

for (i = 0; copyOfP1.getDegree() >= p2.getDegree(); i++) {
    ((CoefficientReal) coefficientsQuotient[i])
    .setCoefficient(((CoefficientReal) copyOfP1.getCoefficients()[0]).
getCoefficient() / ((CoefficientReal) p2.getCoefficients()[0])
.getCoefficient());
    copyOfP1 = subtract(p1, multiply(p2, new Polynomial(StringToPolynomial
.eraseLeadingZeros(coefficientsQuotient, quotientDegree))));
    if (copyOfP1.getDegree() == 0) {
        break;
    }
}

```

Take an example.

$2x^3 + x^2 +$	2	$x^2 + x + 1$
$2x^3 + 2x^2 + 2x$		$2x - 1$
$-x^2 - 2x + 2$		
$-x^2 - x - 1$		
$-x + 3$		

First we know that if the divisor is of the degree 2, then the quotient rest will be of degree at most 1. If the dividend will be of degree 3, then the quotient will have degree $3 - 2 = 1$. In a copyOfP1, we store the first polynomial (p1).

STEP1. We divide the first coefficient of copyOfP1 to the first coefficient of p2. This number will be the first coefficient of the quotient. The following numbers will be next coefficients etc.

STEP2. We assign to CopyOfP1 the subtraction $p1 - \text{multiplication}(p2, \text{quotientThatWeHaveSoFar})$.

STEP3. We repeat the steps above until the degree of copyOfP1 is less than the degree of p2 or copyOfP1 has degree 0.

Now, the quotient has been found. The rest will be computed by the formula:

```
rest = subtract(p1, multiply(p2, quotient));
```

Note that the division method returns an array of polynomials, the quotient (p[0]) and the rest (p[1]).

The **differentiate** algorithm is rather simple. We first create a polynomial with degree smaller with one than of the input polynomial (if the input polynomial hasn't already degree 0) and we assign to each resulted coefficient the input coefficient multiplied by the term's power.

The **integrate** algorithm computes indefinite integrals, but does not show the constant term. First, it creates a polynomial with degree bigger with one than if the input polynomial and assign to each resulted coefficient the input coefficient divided by the term's power plus one.

The **evaluateAt** method is completely reproduced below with no further explanations:

```
public static double evaluateAt(Polynomial p, double n) {
    int i;
```

```

    double pOfN = 0;
    for (i = 0; i <= p.getDegree(); i++) {
        pOfN = pOfN + ((CoefficientInt)
p.getCoefficients()[i]).getCoefficient() * (Math.pow(n, p.getDegree() - i));
    }
    return pOfN;
}

```

The **root finding** method worths taking a look. It uses the bisection method algorithm: take an interval where the functions signs are different at the ends of the interval. It means the function passes through zero at least once along that interval. Now divide the interval in two, and repeat the process recursively as long as the function maintains this property on one of the divided interval. The method will slowly converge to the solution. For converging and testing that the polynomial has real solutions, we have used

```

public static double ERROR = 0.00001, MAXRANGE = 1_000_000;

```

In case the method starts looking for a root in an interval larger than $(x - \text{MAX_RANGE}, x + \text{MAX_RANGE})$ the method throws a `HasNoRealRootException`.

Now switching to the execution of the graphics, we can see the logic is done inside the `GraphFrame` class. The coordinate axis are translated so that to place the origin of the plane in the center of the window. Then, the axis and reference points on the axis are drawn. Then, starting from an initial position, a `GeneralPath` is drawn, with linking points representing the values of the polynomial along the x-axis with a `STEP = 0.01` increment. Of course, we do this by using the `evaluateAt` method from `Functions`, and using a specific method from `GraphFrame` called `excludeOutOfBounds`:

```

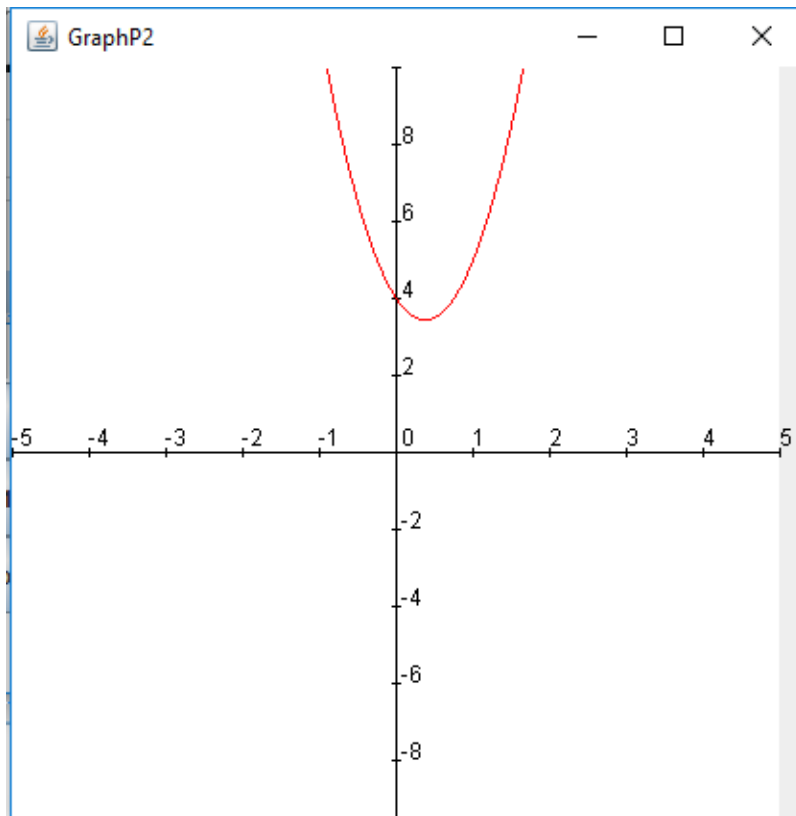
    public double excludeOutOfBounds(double x) {
        while (((Functions.evaluateAt(p, x) * 20 < -200) ||
(Functions.evaluateAt(p, x) * 20 > 200) && (x <= 5))) {
            x += STEP;
        }
        return x;
    }

```


3.8. User Interface

Polynomials					
p1		+	-	*	/
p2		p1'	p2'	int(p1)	int(p2)
x		P1(x)	P2(x)	P1=0 around x	P2=0 around x
result		graph p1	graph p2		

Polynomials					
p1	3 -4 5	+	-	*	/
p2	4 -3 4	p1'	p2'	int(p1)	int(p2)
x	4	P1(x)	P2(x)	P1=0 around x	P2=0 around x
result	37.0	graph p1	graph p2		



4. Implementation and Testing

The project was developed in Eclipse IDE using Java 8, on a Windows 10 platform. It should maintain its portability on any platform that has installed the SDK. It was heavily tested, but new bugs could be discovered in the near future. One of the main inconveniences are that the program is not protected against all kinds of illegal input data, only to some. This could be the subject of future development.

5. Results

The application is a user-friendly, helpful app that can perform basic mathematical operations on integer polynomials of a single variable. It can be extended into a more powerful tool that can handle multiple variable polynomials, double coefficient polynomials and find complex roots. But for now, the app can be considered a good starting point.

6. Conclusions

6.1. What I've Learned

TIME IS PRECIOUS! I had to solve complicated problems of time management which taught me good organization skills. I learned that a good model is always a key to a successful project and that a bad model could ruin your project in the end. I learned never to get stuck on one little bug, and if I do, ask for help, either from the TA or from colleagues

or on different forums. I am looking forward to apply what I have learned in future projects.

6.2. Future Developments

We could try to make the program more user – protected in that we could assume the user doesn't know to enter valid data each time. We could for example show a message if the user presses the graphP1 button but no p1 was provided.

Another improvement could be the use of another algorithm used for computing the root. The bisection method is slow in converging, and we could use methods like Newton's method or, why not, Chebyshev's method.

Also, we could use ArrayList instead of mere arrays for storing Coefficients.

7. Bibliography

- [1] <http://www.uml.org/> accessed 5/3/2016
- [2] Barry Burd, *Java For Dummies*, 2014
- [3] Kathy Sierra, Bert Bates, *Head First Java*, 2005
- [4] Steven Gutz, Matthew Robinson, Pavel Vorobiev, *Up to Speed with Swing*, 1999
- [5] Joshua Bloch, *Effective Java*, 2008
- [6] <http://www.stackoverflow.com/>
- [7] <http://docs.oracle.com/>