Assignment 1 documentation
Polynomial calculator

Litan Virgil

Group 30425

TA :

Contents

1. Functional Requirements 3

2. Objectives 3
   2.1 Main Objective 3
   2.2 Secondary Objectives 3

3. Problem analysis 3

4. Design 3
   4.1 Data structures 3
   4.2 Class diagram 3
   4.3 Algorithms 3

5. Implementation 4

6. Testing 8

7. Conclusions and Further Development 9

8. Bibliography 9

1. Functional Requirements

The application must be able to perform basic unary and binary operations with single variable type polynomials. There are two unary operations that the software implements are: integration and differentiation, while the binary operations are addition, subtraction, multiplication and division.

The polynomials that constitute the input of the calculator should be introduced in the form " 3x ^ 2 + 2x ^ 3 - 2 " .

2. Objectives

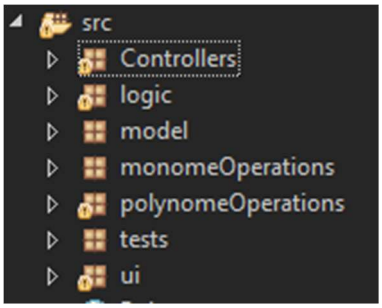
2.1 Main Objective

Designing, developing and testing a software that has a General User Interface and it is capable of processing single variable polynomials of any degree.

2.2 Secondary Objectives

Taking into account some use cases is very useful in order to develop a piece of software. A few of those use cases could be: students that need to exercise their ability to compute polynomials an need to validate their results. Also, there could be people from other fields than mathematics that still need to compute fast various polynomials.

Another objective would be designing the application such a way that it could be easily further developed by separating the models, the general user interface and the logic components and interconnecting them in a single component, namely the " MainController " class which also contains the entry point of the application.



3. Problem analysis

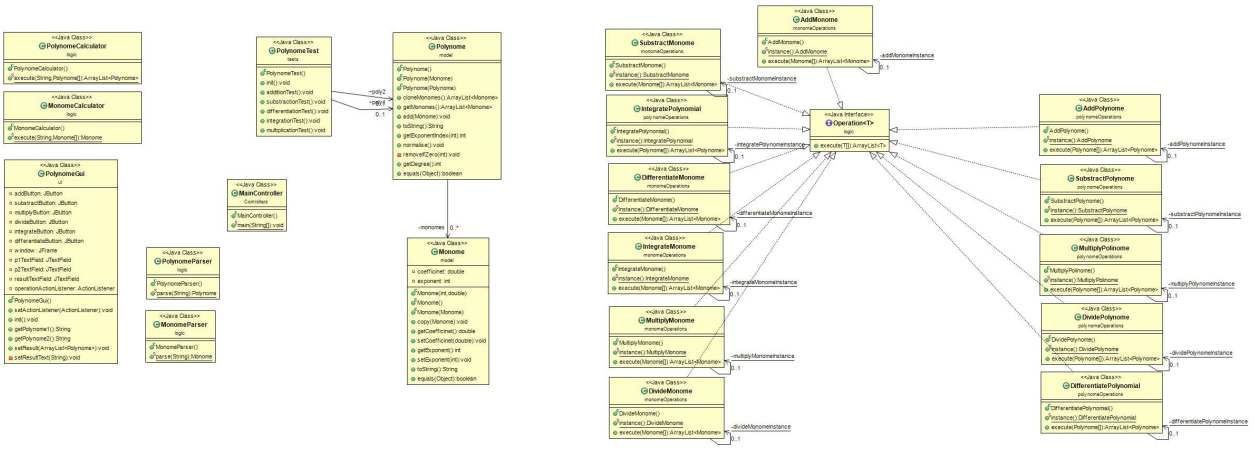
The problem presents itself as being simple enough to be implemented in an imperative manner but also complex enough to be a good candidate for an Object Oriented Programming approach. Thus, an in-depth analysis is highly required. As the concept of polynomials has a well defined meaning in mathematics and other science fields the problem requests close attention on the semantic perspective as well, not just syntactic.

4. Design

4.1 Data Structures

Polynomials are internally represented as an array list of monomials. This approach was considered as a consequence of its obvious advantages, namely: high modularity level and the ease of element management (already implemented by the ArrayList container).

4.2 Class diagram



4.3 Algorithms

Simple algorithms of iterative searching were used in order to find duplicates elements in an array (by means of exponent) or null elements (by means of coefficient). The algorithm with the highest big O function value is the multiplication, with O (n * m) (where n is the number of elements of the first polynomial and m is the number of elements of the second polynomial).

5. Implementation

The models used are the classes Monome and Polynome. The Monome class consists of exponent and coefficient and implements some basic functionality that will prove useful. The main method to be observed is:

```
@Override
public boolean equals(Object o){
    if (o == this) return (true);
    if (o == null) return (false);
    if (!(o instanceof Monome)) return (false);
    Monome m = (Monome)o;
    return ((this.coefficinet == m.getCoefficinet()) && (this.exponent == m.getExponent()));
}
```

Also, the “ Monome ” class does not allow (because of mathetical reasons) to set the exponent of one of its instances with a non-negative value. If this case is encountered the value is discarded.

```
public void setExponent(int exponent) {
    if (exponent < 0) return;
    this.exponent = exponent;
}
```

The Polynome class consists of an array of Monomes and several useful methods. Some of those that are worth mentioning:

```java
//removes 0s, merges duplicates
public void normalise(){
    if (monomes.size() == 0) {
        return;
    }
    for (int i = 0; i < monomes.size(); i++){
        for (int j = i + 1; j < monomes.size(); j++){
            if (monomes.get(i).getExponent() == monomes.get(j).getExponent()){
                monomes.set(i, MonomeCalculator.execute("add", monomes.get(i), monomes.get(j)));
                monomes.remove(j);
            }
        }
        removeIfZero(i);
    }
}
```

As the comments states, this method removes zero elements and merges equal exponent elements, therefor normalizing the polynomial. This class implements also the equals method in order to ensure compatibility with several methods that java provides like assertEquals, that are useful during unit testing.

```java
@Override
public boolean equals(Object m){
    if (m == this) return (true);
    if (m == null) return (false);
    if (!(m instanceof Polynome)) return (false);

    Polynome aux = (Polynome)m;
    if (aux.getMonomes().size() != monomes.size()) return (false);

    for (Monome m1 : monomes){
        if (aux.getExponentIndex(m1.getExponent()) != -1){
            if (aux.getMonomes().get(aux.getExponentIndex(m1.getExponent())).equals(m1) == false){
                return (false);
            }
        }
        else{
            return (false);
        }
    }
    return (true);
}
```

The method firstly considers some trivial cases, then checks the size of the two polynomials. If all those tests pass then the existance and equality of each monomial in the first polynomial is checked with monomials in the second one.

The “ getDegree ” method returns the maximum exponent found on one of the monomes in the list. This method is useful in the case of division operation.

```java
public int getDegree(){
    int max = 0;
    for (Monome monome : monomes){
        if (max < monome.getExponent()){
            max = monome.getExponent();
        }
    }
    return (max);
}
```

Another uuseful method that was added to the “ Polynome ” class was the “ getExponentIndex ” which returns the index of an monome with a given exponent. In case that the monome is not found, the value -1 is returned.

```
public int getExponentIndex(int exponent){
    for (int i = 0; i < monomes.size(); i++){
        if (exponent == monomes.get(i).getExponent()){
            return (i);
        }
    }
    return (-1);
}
```

In order to ensure reusability and modularity, each an “Operation” interface is defined and each operation is implemented in a separate, dedicated class. The return type is an ArrayList data structure to ensure that the

```
public interface Operation<T> {
      ArrayList<T> execute(T...m);
}
```

Interface would suit every type of operation (with one ore more results). Also, the use of generics is justified by the several types of operands (monomials and polynomials). Furthermore, to create a general interface that would suit every type of operation, a variable number of arguments is used (to match operations with 1 operand, like integration and operations with 2 operands, like addition).

Further more, each class was designed considering the singleton design pattern in order to avoid multiple, functionally identical, instances of an operation class.

```
public class AddMonome implements Operation<Monome> {

    private static AddMonome addMonomeInstance; //singleto instance

    public static AddMonome instance(){
        if (addMonomeInstance == null){
            addMonomeInstance = new AddMonome();
        }
        return (addMonomeInstance);
    }

     public ArrayList<Monome> execute(Monome...m) {
        ArrayList<Monome> rez = new ArrayList<Monome>(); ///create a new list of monomes
        Monome mRez = new Monome(m[0]);//instanciate the monome result with the values of the first argument
        if (mRez.getExponent() == m[1].getExponent()){  //if the exponets match
            mRez.setCoefficinet(mRez.getCoefficinet() + m[1].getCoefficinet()); //the coefficints add
        }
        rez.add(mRez); //result is added to the result array
        return (rez);
    }

}
```

Further, using all those components an following the “ factory ” design pattern, a class named MonomeCalculator was created. The method getOperation receives a string as its input and returns an object that implements the Operation <Monome> class. The same manner was used for the PolynomeCalculator class, as shown below.

```
public Operation<Polynome> getOperation(String operation){
    switch (operation){
        case "differentiate":
            return (DifferentiatePolynomial.instance());
        case "integrate":
            return (IntegratePolynomial.instance());
        case "add":
            return (AddPolynome.instance());
        case "substract":
            return (SubstractPolynome.instance());
        case "multiply":
            return (MultiplyPolinome.instance());
        case "divide":
            return (DividePolynome.instance());
    }
    return null;
}
```

The General User Interface was designed to be simple and intuitive. It consists of 2 input fields (instances of JTextField), 1 output label (instance of the class JLabel) and 6 buttons corresponding to the 6 basic operations that the software can perform. As a rule of thumb, in the case of unary operations the first polynomial is used as the operand. All elements were added to a container (JFrame) that forms the main window. In order to convert sequences of characters in instances of the Polynomial class the " parse " was created.

In order to separate the logic and from the general user interface, all buttons have the same actions listener that is set from outside the class.

```
public static Polynome parse(String input) throws Exception{
    Polynome polynome = new Polynome();
    Pattern pattern = Pattern.compile("([+-]?[^-+]+)");
    Matcher matcher = pattern.matcher(input);

    while (matcher.find()) {
        polynome.add(MonomeParser.parse(matcher.group(1)));
    }
    return (polynome);
}
```

This method uses regular expressions in order to identify disjoint groups of characters that are monomials. Further, each group is parsed by a dedicated static method of the MonomeParser.

```java
public static Monome parse(String input) throws Exception{
    Monome monome = new Monome();
    boolean sign = true; //true means positive
    input = input.replaceAll("\\s+", ""); //remove all spaces
    if (input.startsWith("-") || input.startsWith("+")){
        if (input.startsWith("-")){
            sign = false;
        }
        input = input.substring(1);
    }
    int xInd = input.indexOf("x");
    if (xInd == -1){
        monome.setExponent(0);
        monome.setCoefficinet(Double.parseDouble(input));
    }
    else {
        monome.setCoefficinet(Double.parseDouble(input.substring(0, xInd)));
        int pInd = input.indexOf("^");
        if (pInd == -1){
            throw new Exception(" ^ not found in monome");
        }
        else {
            monome.setExponent(Integer.parseInt(input.substring(pInd + 1)));
        }
    }
    if (!sign){
        monome.setCoefficinet(-monome.getCoefficinet());
    }
    return (monome);
}
```

The method returns an instance of the Monome class and gets as input a string. Firstly, all whitespaces are removed. Then the first character is analyzed and “consumed”. Then the “ x ” character is searched. If there exist no “x” the exponent of the monomial is set to 0 and the whole input is parsed as a number. Otherwise the string until the x character is considered to be the coefficient and it is parsed. Further, if the character “ ^ ” is not found in the input string, and exception is thrown, else the substring following the “ ^ ” character is parsed and considered to be the exponent of the resulting monomial.

All software components described so far are used in the MainController class that binds the general user interface with the logic part. This is done by firstly creating a new instance of the PolynomeGUI class and setting the action listener that manages all buttons events. Lastly the gui is initialized and the user starts using the software.

The action listener parses contents of the two text fields and sets the content of two variables that further will consist the operands of the chosen operation to be performed and lastly the result of the operation is

displayed on the label through the setResult method implemented by the general user interface component .

```
public static void main(String[] args) {
    PolynomeGui gui = new PolynomeGui();
    ActionListener operantionListener = new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent e) {
            JButton button = (JButton)e.getSource();
            boolean inputOk = false;
            Polynome p1 = new Polynome();
            Polynome p2 = new Polynome();
            try {
                p1 = PolynomeParser.parse(gui.getPolynome1());
                inputOk = true;
            } catch (Exception e1) {
                System.out.println("error parsing polynome 1 " + e1.getMessage());
            }
            try {
                p2 = PolynomeParser.parse(gui.getPolynome2());
                inputOk &= true;
            } catch (Exception e1) {
                System.out.println("error parsing polynome 2 " + e1.getMessage());
            }
            if (inputOk){
                gui.setResult(PolynomeCalculator.instance().getOperation(button.getText()).execute(p1, p2));
            }
        }
    };
    gui.setActionListener(operantionListener);
    gui.init();
}
```

The text of the button that triggered the event is used as an argument to the getOperation method, returning an object that implements the Operation interface, thus it implements the " execute " method.

6. Testing

Testing was done by using Junit. The dedicated package contains only a single class that implement the testing for polynomial calculator functionality. The test check general use cases and does not include various edge-cases (those must be implemented soon). By using the java style adnotations a more robust set of tests is obtained (the " @Before " adnotation) ensures the consistency of each test.

```
@Before
public void init(){
    try {
        poly1 = PolynomeParser.parse("4*x^1 + 3*x^2");
        poly2 = PolynomeParser.parse("-2*x^1 + 3*x^9");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Test
public void additionTest(){
    try {
        Polynome expected = PolynomeParser.parse("2*x^1 + 3*x^2 + 3*x^9");
        Polynome actual = PolynomeCalculator.instance().getOperation("add").execute(poly1, poly2).get(0);
        assertEquals(expected, actual);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The default values for the general test case are "4*x^1 + 3*x^2" and "-2*x^1 + 3*x^9". While addition is expected to output the polynomial 2*x^1 + 3*x^2 + 3*x^9 and the multiplication operation the polynomial: -8*x^2 + 12*x^10 - 6*x^3 + 9*x^11, thus basic adding and merging monomials inside a polynomial is tested.

7. Conclusions and further developing

In conclusion, developing a piece of software that would perform basic operations on polynomials is a process that requires attention from the technical point of view but also from the semantic point of view, (ensuring that the results actually make sense for a mathematician). Also, an object oriented approach is very prolific because of the high modularity that this kind of approach confers.

First improvement would be implementing the " execute " method of the DividePolynome class. Further developing of this program would focus on developing more tests that would cover all mathematical edge cases as well as strange input and the error handling mechanism of the code. Also, to ensure better consistency and robustness the " parse " method of the MonomeParser class should be enhanced in order to accept monomials without the " ^ ".

In order to extend the functionality of the program the capability to execute more operations like computing the value in a point or computing the integral on a user – defined interval would be needed. Also, a useful feature would be the " re-use " functionality, that would let the user reuse the result as the first or second operand, without coping and pasting. Also, the " swap " button that would swap the two polynomials would be useful in the case of division and subtraction.

8. Bibliography

http://stackoverflow.com/questions/36490757/regex-for-polynomial-expression
http://www.journaldev.com/1827/java-design-patterns-example-tutorial