Student: Gergő Papp-Szentannai
Group: 30421

Programming Techniques 2016

Documentation

Polynomial Processing

Homework number: 1
Due date: week 4

1. Purpose

The purpose of the project is to design and implement a program for processing polynomials. The main aspect is respecting the rules of OOP design, using abstraction, encapsulation, inheritance and polymorphism, while making sure that the whole program has a low coupling and high cohesion. In other words, one class should represent one and only one abstracted object and the methods should only have one well defined purpose. The design of the program should be as clear and logical as possible even if this means a slight trade-off with performance.

The purpose of this project is, in the first place, to create a safe, modular and easy-to-reuse program, which might also be further developed without any headaches.

2. Problem analysis

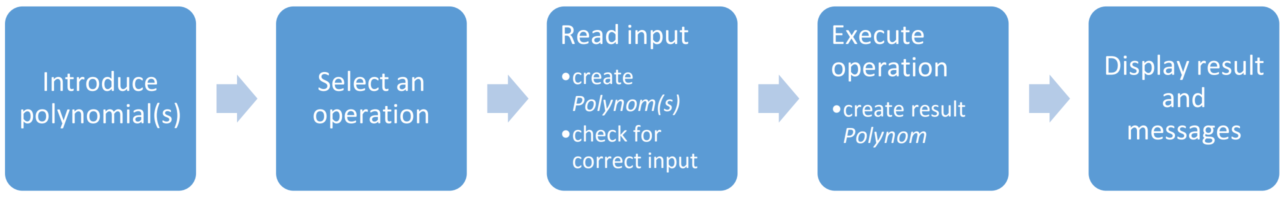
<Modelling, scenarios, usage>

To see how the program should behave, we have to look at real life scenarios. Generally, we use polynomials in physics, mathematics and engineering, because they model something – this can be any data, for example the acceleration of a car. To be able to use this data we need a data structure that can store any polynomial.

We also need to manipulate these polynomials, so we need to implement mathematical functions between them. These can be addition, subtraction, etc. Derivation and integration are also important operators, which should be implemented.

After we store and also manipulate the given data, we also need to make it available for the users, so we need to implement a simple but powerful graphical user interface.

A success scenario, based on the already mentioned model is presented in the following use case diagram:



If there are errors (eg. the user did not type in a correct form), a message will be displayed, informing the user.

To summarize the problem analysis, we have to design three separate parts: storing the polynomials, applying operations on them and interacting with the user. These have to be separate, well defined parts that do not depend on each other's implementation. The next part focuses on how to design these three main parts and how to link them efficiently.

3. Design

<UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface>

Data structure

Because it is hard to manipulate a whole polynomial at once, we have to break down the problem to smaller parts. We can specify a polynomial as a collection of "monoms". Another solution would be to simply specify the polynomial as an array (or ArrayList), where the position/index of an element corresponds to the power of that element. At first this might seem a good method, because we could gain performance. Mathematical operations would also be faster – we could manipulate polynomials as vectors and matrices. However, from another perspective, this would mean that the presented method mainly depends on the hardware implementation – which we do not want.

Instead we have to make an abstraction from the real world: a polynomial has a positive number of values in any possible order (eg. $5x^2 - 3x^0 + 1x^1$). These elements should be independent of each other. This is why the most appropriate data structure for storing a polynomial is the following: we specify a relationship between a polynomial ("polynom") and different monoms. This way a polynomial has a monom (or more) – see Figure 1. Each monom stores only a value and the power to which it corresponds. We also have to make sure that there are no two different monoms having the same power, corresponding to the same polynomial. The UML diagram of the polynomial as a data structure is shown below – we can also notice the has-a relationship:

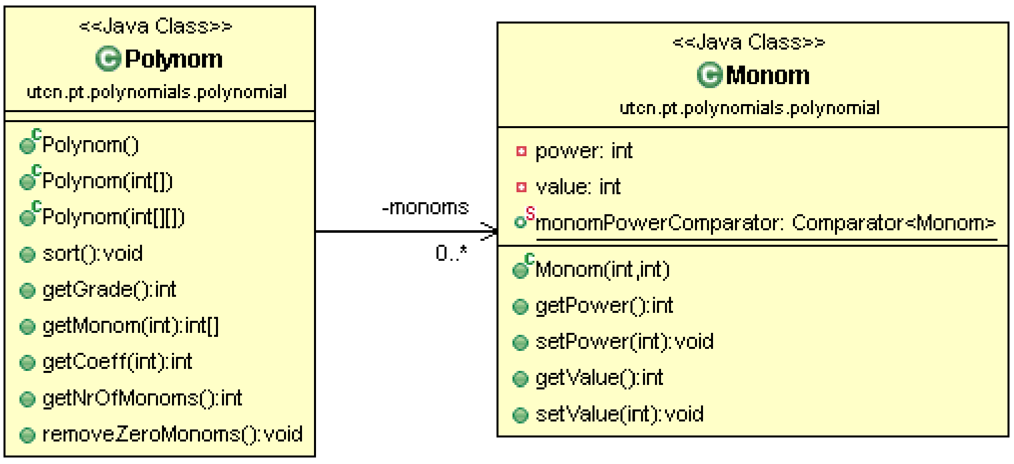


Figure 1

Operators

To manipulate the polynomials, we need one or more polynomials and an operator. We can notice from mathematics that the most often used operators are: addition, subtraction, multiplication and division. Derivation and integration are also widely used. We can group these operators as unary and binary, depending on how many inputs we need. The output of these operators is always one polynomial. To make the polynomial processing easy to implement and to further expand, I create the *Operator* interface which is extended by the *BinaryOperator* and *UnaryOperator* interfaces. These interfaces contain the *compute()* method which takes as parameter one or to polynomials and also returns one polynomial.

Each operation represents a separate class – each of these classes implement the *compute()* method in a different way. The class diagram of the operations is represented in Figure 2. Each of these operators create a new instance of the class *Polynom*, which they return to the caller. It is important to notice that none of the operators depend on the implementation of the polynomials. Instead they use getters and setters (eg. *getCoeff(int power)*). This way the program remains modular – one part does not depend on the implementation of another part.

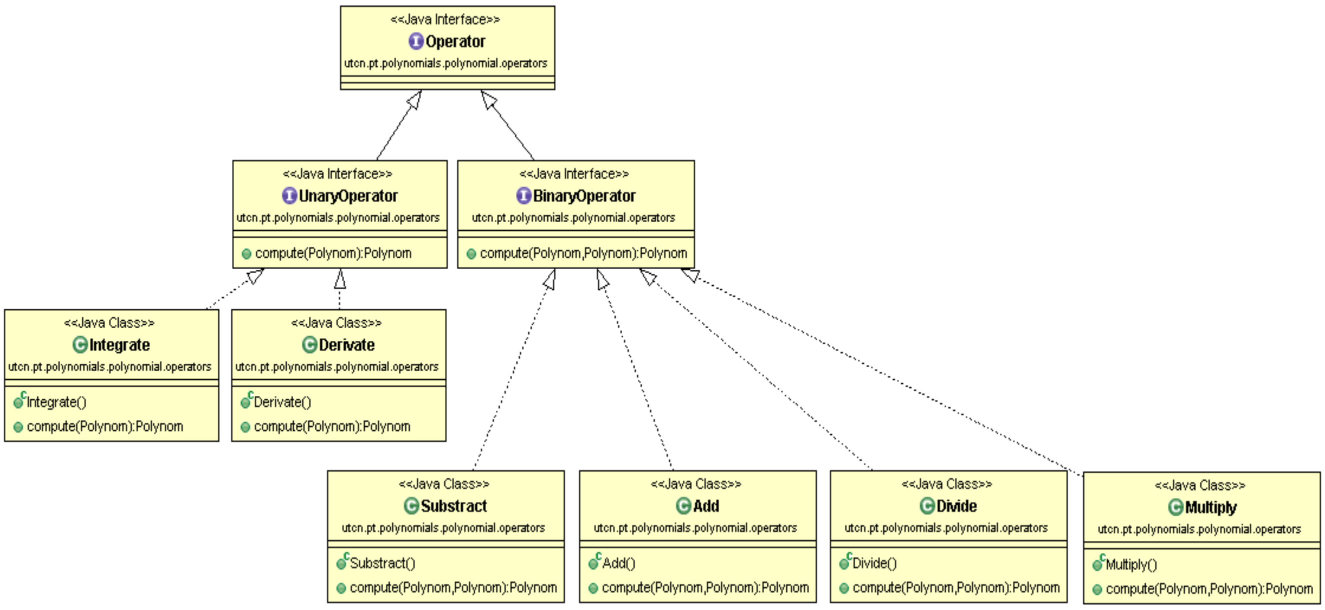

<<Java Interface>>
Operator
utcn.pt.polynomials.polynomial.operators
<<Java Interface>>
UnaryOperator
utcn.pt.polynomials.polynomial.operators
compute(Polynom):Polynom
<<Java Interface>>
BinaryOperator
utcn.pt.polynomials.polynomial.operators
compute(Polynom,Polynom):Polynom
<<Java Class>>
Integrate
utcn.pt.polynomials.polynomial.operators
Integrate()
compute(Polynom):Polynom
<<Java Class>>
Derivate
utcn.pt.polynomials.polynomial.operators
Derivate()
compute(Polynom):Polynom
<<Java Class>>
Substract
utcn.pt.polynomials.polynomial.operators
Substract()
compute(Polynom,Polynom):Polynom
<<Java Class>>
Add
utcn.pt.polynomials.polynomial.operators
Add()
compute(Polynom,Polynom):Polynom
<<Java Class>>
Divide
utcn.pt.polynomials.polynomial.operators
Divide()
compute(Polynom,Polynom):Polynom
<<Java Class>>
Multiply
utcn.pt.polynomials.polynomial.operators
Multiply()
compute(Polynom,Polynom):Polynom

Figure 2

User interface

To design a usable program, we also need to design a good user interface. It has to be as simple as possible but it also has to be complex enough for the user to compute various operations. I decided to have a single frame for my application, which will be split in 5 equal regions:

1. Polynomial input A
2. Polynomial input B
3. Operators (a separate button for each operator)
4. Result
5. Status messages and an additional button

The user will input the polynomials in the fields A and B, then select an operator. The result will be displayed on the Result field. Any important messages (warnings, error messages and the status of the current operation) will be displayed on the last/bottom field – see Figure 3.

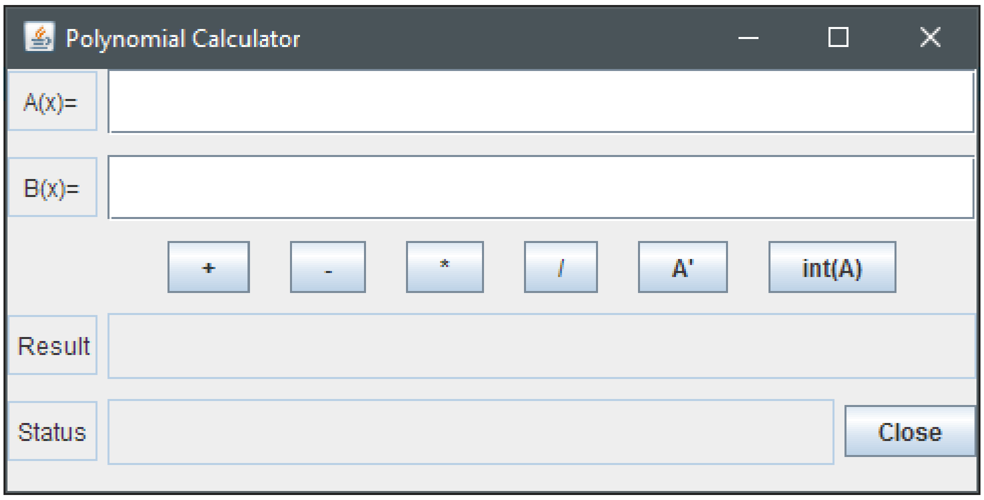


Figure 3

The five parts are organized in a vertical box layout. Each part is a JPanel, arranged also in a box layout, but using the horizontal setting. The frame can be resized by the user and the panels will still be correctly organized because the components are "glued" together. Between the components there are some "rigid areas" which add more space.

4. Implementation and testing

Implementation

The entry point of the program is the *main()* method inside the *App* class, where I instantiate the *MainFrame* class in a thread. *MainFrame* contains all the elements used by the GUI, including the action listeners for each button, and two extensions of the *Operator* interface – *BinaryOperator* and *UnaryOperator*. When a button (operation) is pressed, the corresponding *Operator* (unary or binary) is instantiated as one of the following classes:

- Add
- Subtract
- Multiply
- Divide
- Derivate
- Integrate

This is followed by calling the *unaryCompute()* or *binaryCompute()* methods which calls the *compute()* method implemented by each class. The resulting *Polynom* is then printed in the Result field. As an advantage each operation is called in the same way – we can easily add new operators and features. The *Operator* classes need to access the input polynomial's coefficients/powers (*Monoms*). This is done through different getters, such that the operations do not depend on the implementation of the *Polynom* class.

The implementation of the Polynom class is mostly presented at the data structure part, however there are some extra helper functions, as presented below:

- *void sort()*: organizes the *Monom*s based on the power, in ascending order. Mostly used for printing to the user in an easier-to-read form;
- *int getGrade()*: returns the highest power of the polynomial with non-zero coefficient;
- *int[] getMonom(int index)*: returns the indexth monom as an array ({*power*, *coefficient*});
- *int getCoeff(int power)*: returns the coefficient at the specified power. Returns 0 if no such power found;
- *void removeZeroMonoms()*: removes monoms having 0 as coefficient. This is especially useful after derivation and subtraction.

These methods are mostly used by the operators. The implementation of these operators simply follow the rules of mathematics – they are done monom-by-monom, just as a normal user would compute polynomials on a paper. For detailed information about operators see the Operator Implementation section.

To make the interconnection between the user and the computer I've included the polynomials.io package, which manages reading and writing to the graphical user interface. The package contains the Printer interface, having two methods (*printPolynom()* and *printStatus()*). These methods are implemented by the *UIPrinter* class, which displays content on the GUI. By using an interface I allow different implementations for printing. For example, one can implement a different class for writing to the console, a file, or even a database. The *UIReader* is also part of the package and it reads the polynomial introduced by the user as a String, converts it to coefficients and then instantiates a new *Polynom* using the acquired coefficients. I used the *split()* method to break the string in substrings and extract the values.

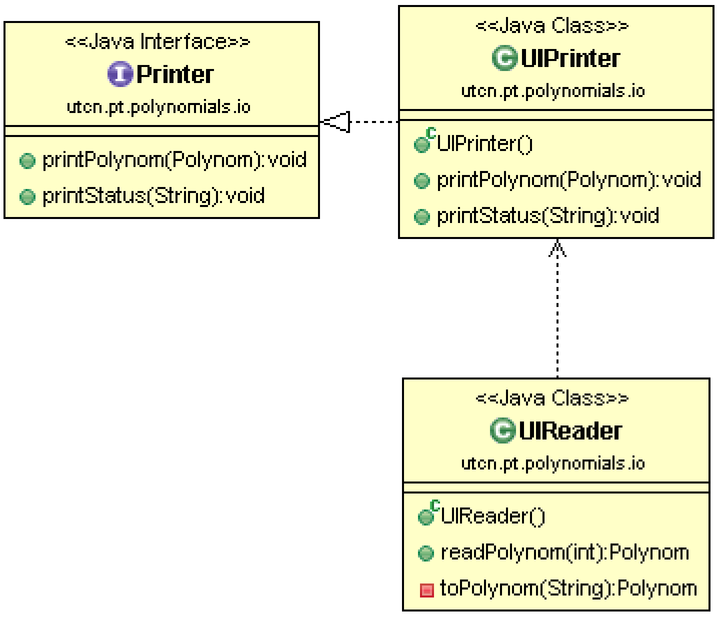


Figure 4

As you can also see in Figure 4, the *readPolynom()* method has a parameter. This helps to specify from where to read the polynomial. The values can be:

- 0 – read from field A
- 1 – read from field B

Other values are not (yet) permitted, however we can easily add new implementations, for example reading form a file or from a website.

Operator Implementation

To show an example I will present the Addition operator.

`public Polynom compute(Polynom A, Polynom B) {`	➢ A and B are the input polynomials
`    int i;`	
`    int max = max(A.getGrade(), B.getGrade())+1;`	➢ Get the grade of the Result polynomial
`    int coeff[] = new int[max];`	➢ Initialize a vector for the Result coefficients
`    for (i = 0; i < max; i++) {`	
`        coeff[i] = A.getCoeff(i) + B.getCoeff(i);`	➢ Add the coefficients term-by-term
`    }`	
`    Polynom P = new Polynom(coeff);`	➢ Instantiate a new polynomial having the new coefficients
`    P.removeZeroMonoms();`	➢ Remove elements which have 0 as coefficient
`    return P;`	➢ Return the Result polynomial
`}`	

You can notice, that the *compute()* method does not directly access the coefficients of the polynomial; instead getter methods are used. If the implementation of the polynomial, as a data structure, would change, the operation would still work – provided that the getters are returning data in the same way.

The other operators behave in a similar way, but the new coefficient are computed according to the needed result. For example, when multiplying, I combine each element from A with each element from B. The powers are added and the coefficients multiplied. In the case of the unary operators the implementation is even simpler – each new monom depends on only one monom from the input polynomial.

For more details, see the source code.

Testing

For testing the program in early stages I added a *test()* method, which reads and displays polynomials. Some of these polynomials have errors and/or mistyped parts. In these cases, I test how the program handles the errors – an error message has to be printed to the user.

In later stages, where the GUI was almost complete, I tested by directly entering polynomials and executing different operations.

5. Results

See the full class diagram at the last page.

I succesfuly designed and implemented the given problem using OOP design. In my program I implemented the following requirements:

- Incapsulation
- Small classes (except for the MainFrame class)
- Small methods – they do one and ony one well defined task
- Java naming conventions, as well as comments and Javadoc comments
- GUI
- Operations:
  - Addition
  - Substraction
  - Multiplication – still needs improvements, but works in all cases
  - Derivation
  - Integration
- Good OOP design
  - Low coupling & high cohesion
  - Usage of interfaces and extended classes
  - High modularity – easy to extend the program

6. Conclusions

<What have I learned, further improvements & development>

I realized that spending a lot more time on properly analyzing and designing a problem can, on the long term, bring great advantages. We do not only save time at later stages of development but we can also easier extend or change some parts of the project without affecting the rest of the program. Initially – as most people – I planned to implement the polynomial as an array and I'm pretty sure implementing some of the operations would have been a lot easier. However, completely different parts would have depended on each other's implementation. This would have led to difficulties as the complexity increased.

7. Bibliography

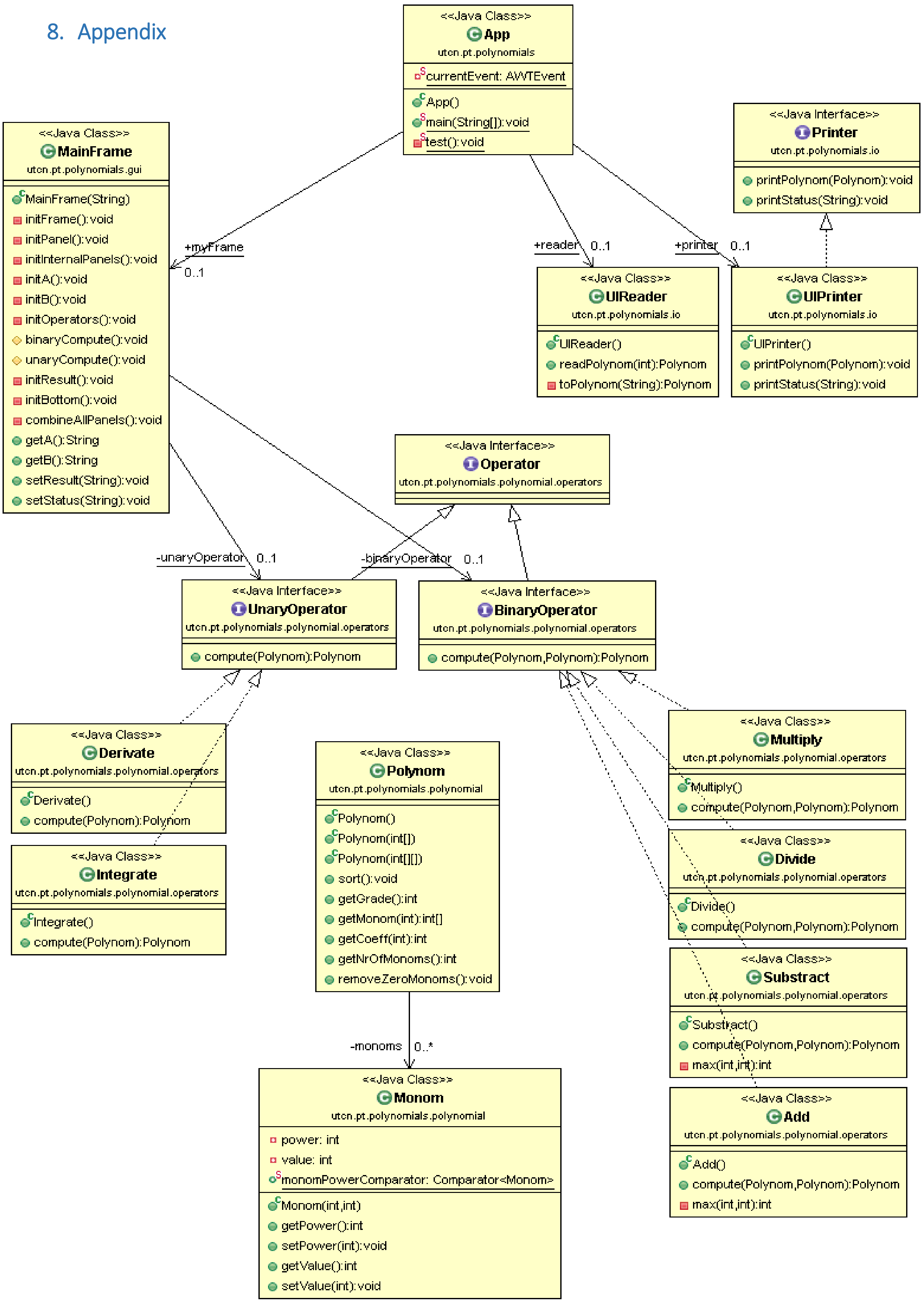
1. PT-Laboratory-1.pdf (http://coned.utcluj.ro/~marcel99/PT_2016/PT-Laboratory-1.pdf)
2. PT-Laboratory-2.pdf (http://coned.utcluj.ro/~marcel99/PT_2016/PT-Laboratory-2.pdf)
3. http://zetcode.com/tutorials/javaswingtutorial/swinglayoutmanagement/
4. http://docs.oracle.com/javase/tutorial/uiswing/layout/box.html
5. http://stackoverflow.com/questions/18405660/how-to-set-component-size-inside-container-with-boxlayout
6. http://stackoverflow.com/questions/3481828/how-to-split-a-string-in-java
7. http://www.tutorialspoint.com/java/java_string_split.htm
8. http://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Contents

The source code, together with the documentation and additional diagrams, is available on GitHub: https://github.com/gergo13/PT2016_30421_Papp-Szentannai_Gego/tree/master/polynomials.

Please note, that the actual code and graphical user interface might suffer small modifications that may or may not be specified in this documentation.

8. Appendix


<<Java Class>>
App
utcn.pt.polynomials
currentEvent: AWTEvent
App()
main(String[]):void
test():void
<<Java Class>>
MainFrame
utcn.pt.polynomials.gui
MainFrame(String)
initFrame():void
initPanel():void
initInternalPanels():void
initA():void
initB():void
initOperators():void
binaryCompute():void
unaryCompute():void
initResult():void
initBottom():void
combineAllPanels():void
getA():String
getB():String
setResult(String):void
setStatus(String):void
+myFrame
0..1
+reader
0..1
+printer
0..1
<<Java Interface>>
Printer
utcn.pt.polynomials.io
printPolynom(Polynom):void
printStatus(String):void
<<Java Class>>
UIReader
utcn.pt.polynomials.io
UIReader()
readPolynom(int):Polynom
toPolynom(String):Polynom
<<Java Class>>
UIPrinter
utcn.pt.polynomials.io
UIPrinter()
printPolynom(Polynom):void
printStatus(String):void
<<Java Interface>>
Operator
utcn.pt.polynomials.polynomial.operators
-unaryOperator
0..1
-binaryOperator
0..1
<<Java Interface>>
UnaryOperator
utcn.pt.polynomials.polynomial.operators
compute(Polynom):Polynom
<<Java Interface>>
BinaryOperator
utcn.pt.polynomials.polynomial.operators
compute(Polynom,Polynom):Polynom
<<Java Class>>
Derivate
utcn.pt.polynomials.polynomial.operators
Derivate()
compute(Polynom):Polynom
<<Java Class>>
Integrate
utcn.pt.polynomials.polynomial.operators
Integrate()
compute(Polynom):Polynom
<<Java Class>>
Polynom
utcn.pt.polynomials.polynomial
Polynom()
Polynom(int[])
Polynom(int[][])
sort():void
getGrade():int
getMonom(int):int[]
getCoeff(int):int
getNrOfMonoms():int
removeZeroMonoms():void
-monoms
0..*
<<Java Class>>
Monom
utcn.pt.polynomials.polynomial
power: int
value: int
monomPowerComparator: Comparator<Monom>
Monom(int,int)
getPower():int
setPower(int):void
getValue():int
setValue(int):void
<<Java Class>>
Multiply
utcn.pt.polynomials.polynomial.operators
Multiply()
compute(Polynom,Polynom):Polynom
<<Java Class>>
Divide
utcn.pt.polynomials.polynomial.operators
Divide()
compute(Polynom,Polynom):Polynom
<<Java Class>>
Substract
utcn.pt.polynomials.polynomial.operators
Substract()
compute(Polynom,Polynom):Polynom
max(int,int):int
<<Java Class>>
Add
utcn.pt.polynomials.polynomial.operators
Add()
compute(Polynom,Polynom):Polynom
max(int,int):int