

POLYNOMIAL CALCULATOR DOCUMENTATION

(ASSIGNMENT 1)

Student Name: Cristea Tudor

Group: 30423

CONTENT

1. ASSIGNMENT OBJECTIVE ………………………………………………………... 3
2. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES …………...….. 3
3. DESIGN AND IMPLEMENTATION ……………………………………………..… 4
4. RESULTS ……………………………………………………………………………. 8
5. CONCLUSIONS …………………………………………………………………….. 8
6. BIBLIOGRAPHY ……………………………………………………………….…… 9
7. ASSIGNMENT OBJECTIVE

The objective of this assignment is to design and implement a polynomial calculator with a dedicated graphical interface through which the user can insert polynomials, select the mathematical operation (i.e., addition, subtraction, multiplication, division, differentiation and integration) to be performed and obtain the result.

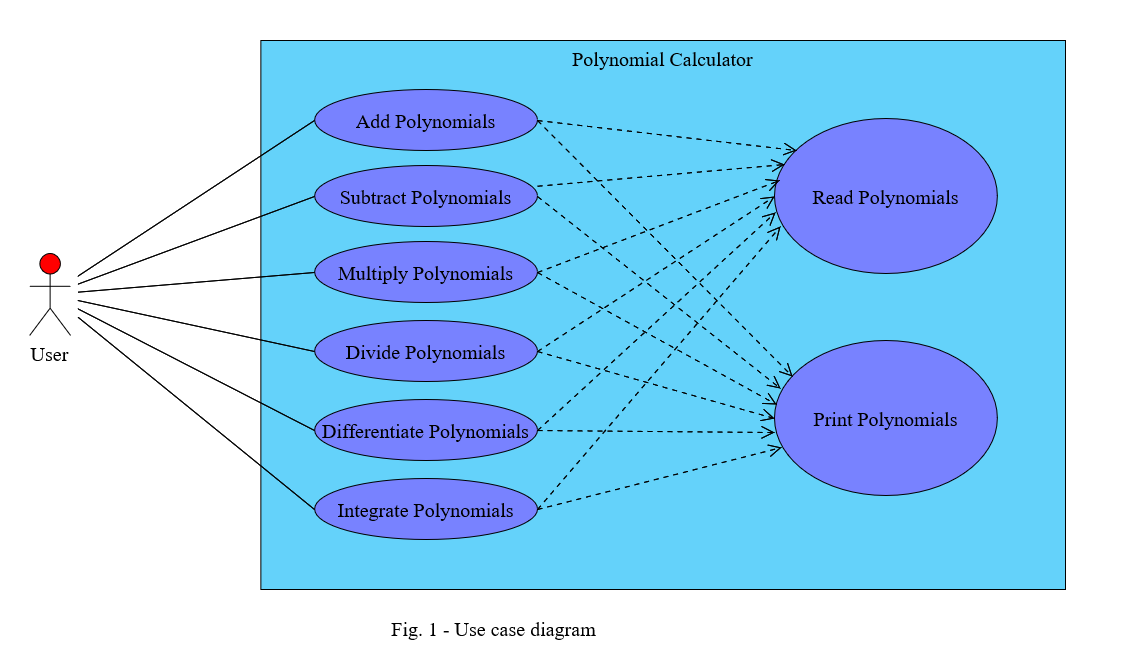
**Note**: Consider the polynomials of one variable and floating-point coefficients.

The sub-objectives are the following:

* + - * Design and implement the *Polynomial* class
* Design and implement operations for reading and printing a polynomial
* Design and implement each operation to be performed on polynomials: addition, subtraction, multiplication, division, differentiation and integration
* Design and implement the Graphical User Interface (GUI)
* Design and implement the controller that links the models (*Polynomial*) to the view (the interface)
* Testing and Debugging possible errors and/or edge-cases

1. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

The use case diagram is depicted in Figure 1. Every operation that is performed on the two polynomials depends on reading (and converting) them correctly as well as printing them on the screen.



**Use case: Add/Subtract/Multiply/Divide/Differentiate/Integrate Polynomials**

**Primary actor: User**

**Success Scenario Steps:**

1. The user enters the first polynomial into the calculator (in the designated text field and by obeying the rules of notation that are specified)
2. The user enters the second polynomial into the calculator (in the designated text field and by obeying the rules of notation that are specified)
3. The user presses the “*Add/Subtract/Multiply/Divide/Differentiate/Integrate Polynomials*” button
4. The two strings are converted internally to polynomials
5. The selected operation is performed using the two polynomials
6. The resulted polynomial is converted to a string
7. The resulted string is displayed in the result text field

**Alternative Scenarios:**

1. The user enters an invalid polynomial in one (or both) text fields (does not obey the aforementioned rules):
   * + The application informs the user that the text he entered is invalid
     + No operation can be performed until the polynomials are valid
     + The scenario returns to step 1 or step 2
2. The user leaves at least one text field empty

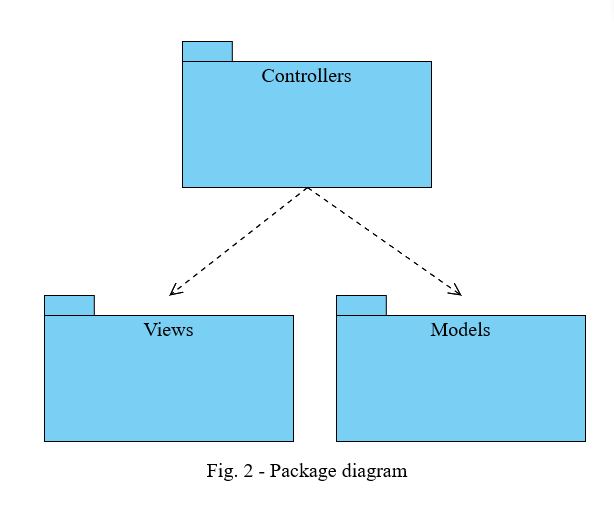
* The application informs the user that there is at least one empty string
* No operation can be performed until both text fields are filled (correctly)
* The scenario returns to step 1 or step 2

1. The user enters the polynomial “*0*” (zero) in at least one of the text fields and presses the “*Divide Polynomials*” button

* The application informs the user that division by zero is impossible
* The division operation cannot be performed until there are no more zeros in any of the text fields
* The scenario returns to step 1, step 2 or step 3 (if the user wishes to perform an operation different from division)

1. DESIGN AND IMPLEMENTATION

The package diagram is depicted in Figure 2. I decided to group my classes based on their functionality, and thus, adopted the MVC (Models, Views, Controllers) pattern for developing this application. This way, the application has a high coherence.



The *Polynomial* class diagram is depicted in Figure 3. In order to make the application as efficient as possible when it comes to the various operations that need to be performed on polynomials, I need to use a certain data structure that has an average case complexity of *O(1)* (constant time) for inserting and selecting any monomial from the polynomial. Knowing this, I chose to use a HashMap data structure, where the keys are represented by the degrees of the monomials (which are of type *Integer*) while the values are represented by the coefficient of the monomials (which are of type *Double*).

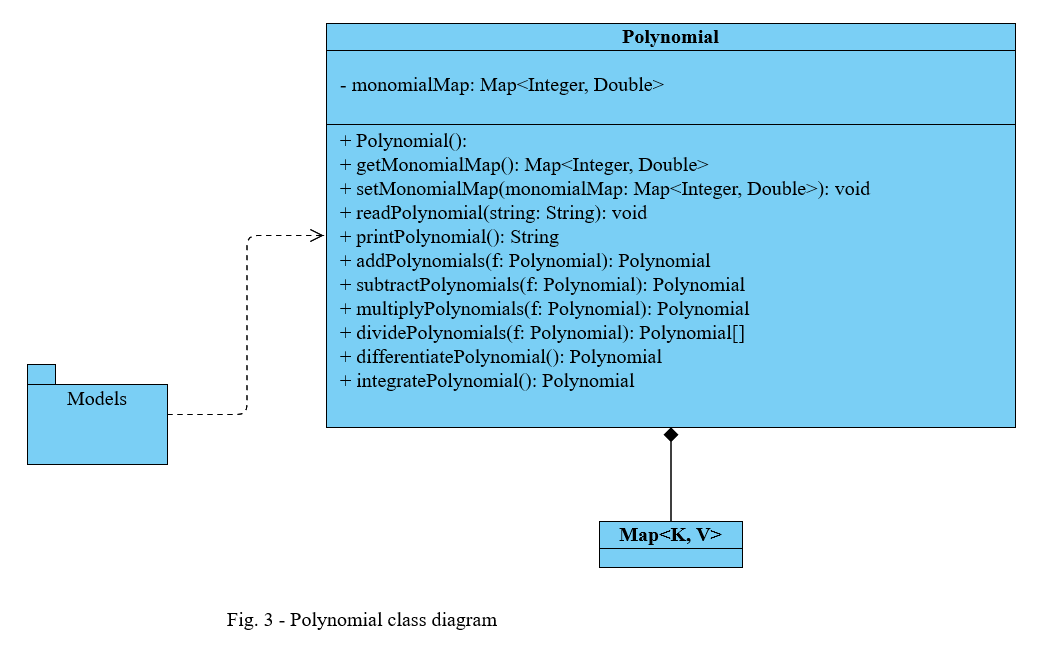
The *readPolynomial()* method uses a regular expression in order to validate that the given polynomial has the correct structure and that the rules of notation were followed. If they were not followed, then the user will receive a prompt with a message that indicates the source of the problem. If the text string is validated, then it is split by using two other regular expressions: one that splits the polynomial into monomials and another that splits the monomial into coefficient and degree. This way, each individual monomial can be easily and efficiently added to the map of the polynomial.

The *printPolynomial()* method does the opposite of the *readPolynomial()* method, meaning that it converts a polynomial, which is composed of a map of its monomials to a string of characters that can be displayed on the screen. However, in order to display it in a pleasant way, the monomials are displayed in a descending order by their degree. In order to do that, we copy the contents of the polynomial’s *HashMap* into a reverse ordered *TreeMap*, since the elements that are inside this data structure, unlike all the other *Map*s, have a certain order. Another important aspect of this method is related to the rational coefficients. If the integer value corresponds to the rational value of a coefficient, then the integer value will be displayed. If it does not, then the rational value will be displayed instead, but only containing the first two decimals of the number (the number will be rounded to those two decimals, not truncated).

For the *addPolynomials()*, *subtractPolynomials()* and *multplyPolynomials()* methods, an essential aspect is to regularly check whether one of the monomials of the result has a coefficient equal to zero, case in which it does not need to be displayed anymore. For the *dividePolynomials()* method, the algorithm of “Long Division” is used in order to obtain the quotient and the remainder of the operation. It consists of the following steps:

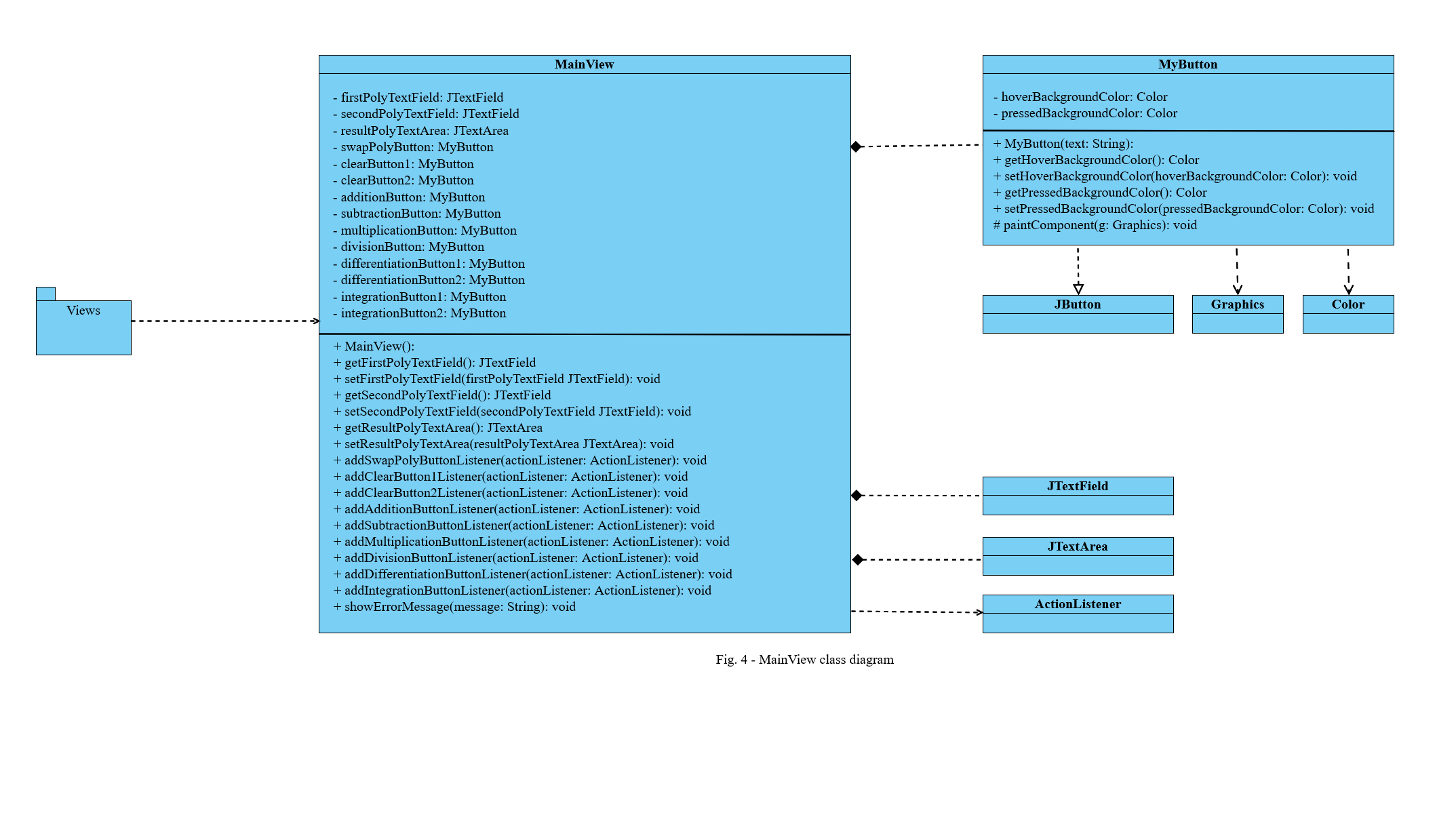
1. The monomials of the two polynomials F and G are ordered descendingly according to their degree
2. The polynomial with the higher degree needs to be divided to the polynomial having the lower degree (let’s consider that F has the higher degree out of the two; if G has the higher degree, then two polynomials are swapped)
3. The first monomial of F is divided to the first monomial of G, and thus, the first term of the quotient is obtained
4. The quotient is multiplied with G and the result is subtracted from F; the result of the subtraction represents the remainder of the division
5. The procedure is repeated from step 2, considering the remainder as the new dividend (the new F) of the division, until the degree of the remainder is lower than the degree of G

All methods that represent the operations performed on polynomials have only one parameter, because the other parameter is considered, by default, to be the polynomial that called the method. However, the result of any of these methods is returned and therefore, can be kept in a separate variable.

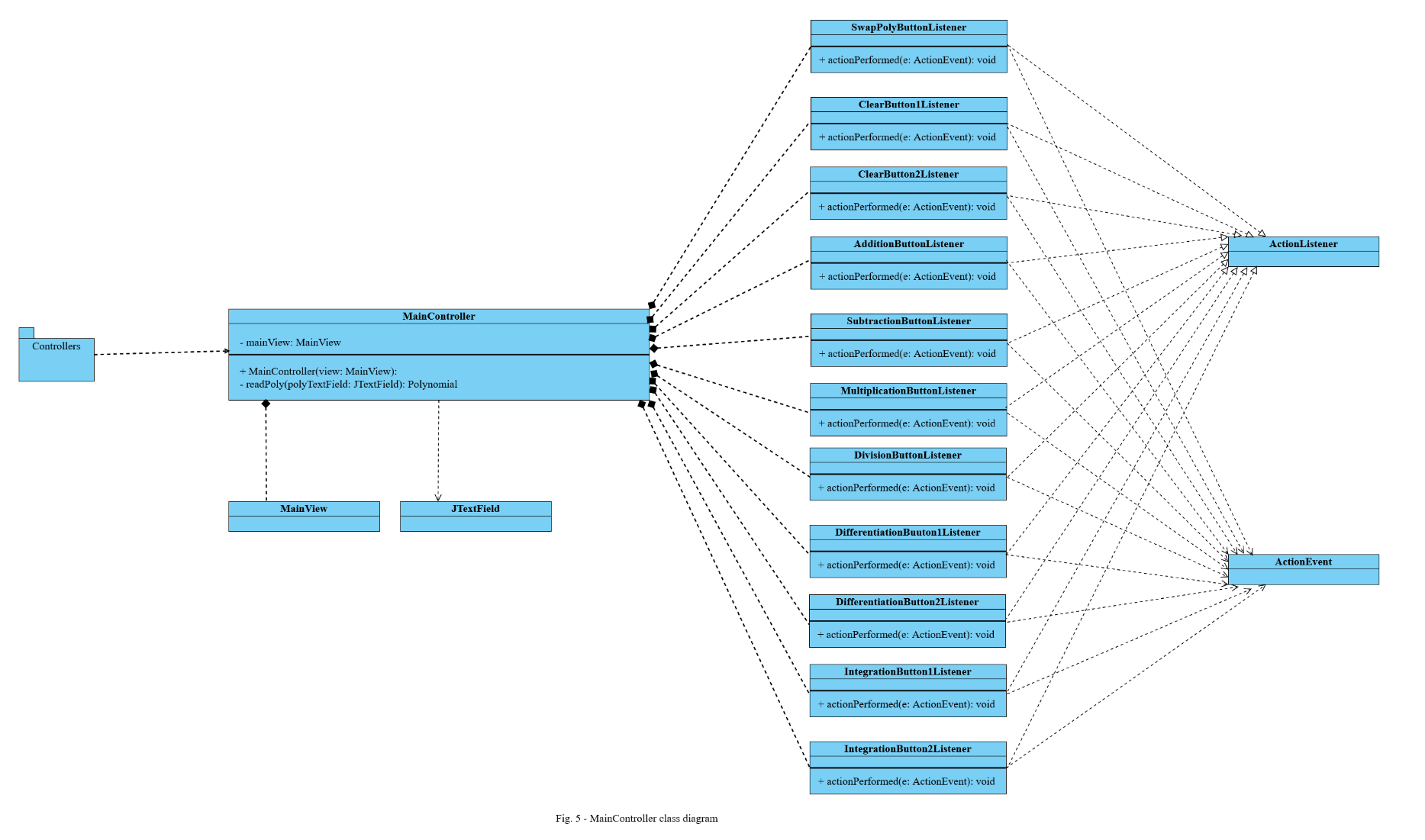


The *MainView* class diagram is depicted in Figure 4. I chose a *GridLayout* for this view since it is responsive to window resizing and it is the most appropriate for an interface which contains a lot of buttons. The buttons that are used, are a subclass of the *JButton* class with the added functionality that they have different colors for when the cursor is not over them, when the cursor hovers over them and when they are pressed. All user interactable elements of the view are declared as fields of the class, while the others are declared only inside the constructor of the class. Besides the buttons that are used to trigger the operations on polynomials, that are some other buttons that have been added in order to make the user’s experience of using the application more efficient and convenient: *Clear* buttons for each polynomial so that the user can easily delete the contents of any of the two text fields as well as a *Swap Polynomials* button so that the user can easily swap the two polynomials and obtain the result that they want from any of the operations (if it is possible). In addition, there is *Note* section where the format of the polynomials is explained.

The *addButtonListener()* methods are implemented in order to add action listeners on the buttons of the interface so that they can do what they are supposed to do. The *showErrorMessage()* method will instantiate a *JOptionPane* with the appropriate message, prompting the user to fix the error that they created, whether they left at least one of the text fields empty, did not respect the rules of the format of the polynomials or tried to divide by zero.



The *MainController* class diagram is depicted in Figure 5. This class has multiple inner classes that implement the *ActionListener* interface and override the *actionPerformed()* method. Inside most of these methods, the first instruction that is executed is a call to the *readPoly()* method which is responsible with calling the *readPolynomial()* method of the *Polynomial* class and, in case there is an error, call the *showErrorMessage()* method of the *MainView* class. Other than that, the sequence of instructions is straightforward: the strings are extracted from the two polynomial text fields; they are converted to actual polynomials; the operation is performed using the two polynomials; the result is converted back to a string and displayed in the result text field of the view. In addition, the constructor of this class is responsible with calling all *addButtonListener()* methods of the view using an instance of the respective inner class as the parameter.



1. RESULTS

The application was tested using the JUnit testing method. Every method, including addition, subtraction, multiplication, division, differentiation, integration in addition to reading and printing of polynomials was unit tested by using some usual polynomials as well as some special cases, in order to cover all possible scenarios. For the six mathematical operations, I have tested the following cases:

* a few regular polynomials (nothing special)
* polynomials such that at least one of the monomials of the result vanishes (because the coefficient becomes zero)
* both polynomials have a degree of zero, so they are represented only by a rational or integer number
* one of the polynomials is zero, the other is different from zero
* both polynomials are zero

For the last two cases, the division operation will throw an exception. For the reading and printing operations I have tested the following cases:

* a few strings of characters that represent regular polynomials (nothing special)
* a string of characters that respects the format but has a lot of whitespace inside of it
* a string of characters which contains only a rational or integer number (so there is no *x* variable, but it still represents a valid polynomial)
* an empty string
* a string which contains only the *“0”* character
* a few strings of characters that do not respect the format, i.e., they either have illegal characters (for example, *“a”* or *“b”*) or one of the monomials has a negative degree (only for reading)
* a few strings of characters which contain several monomials of the same degree (only for reading)

All tests were passed and the coverage for every method was equal to 100%.

1. CONCLUSIONS

In conclusion, a polynomial calculator is a very useful tool for people that use polynomials frequently, such as engineers, mathematicians, students or even programmers, because the result is obtained instantly and they do not need to do the computations themselves (by hand), or it can be used only to verify the results of their own computations.

Working on this assignment, personally I have learned how to use the *Map* interface as well as its *TreeMap* and *HashMap* implementations. I have learned more about regular expressions, how to construct them and how to integrate them in my program in order to validate data. I have learned that data validation is really important for any program that interacts with a random user, because if the flow of data is not monitored, then numerous and various unpleasant errors may occur. I have learned how to use JUnit in order to unit test the methods that I write and I have discovered that although it is more time consuming because I have to write more code, it is easier to add more test cases or modify existing ones. Lastly, I have learned how to make the buttons of the interface have different colors based on the position and the action of the mouse.

Obviously, the application can be developed further with additional features, such as:

* computing the roots of a polynomial (only if the degree is smaller than five)
* allowing the polynomials to have negative degrees and complex coefficients
* allowing polynomials of two or more variables
* evaluating the polynomial at a specific point
* plotting the graphical representation of a polynomial

1. BIBLIOGRAPHY

* [online.visual-paradigm.com](https://online.visual-paradigm.com) (for diagrams)
* *Fundamental Programming Techniques* course lecture slides and laboratory slides
* [stackoverflow.com](https://stackoverflow.com)
* [chat.openai.com/chat](https://chat.openai.com/chat)
* [geeksforgeeks.org/](http://www.geeksforgeeks.org/)
* [youtube.com](https://www.youtube.com) (for tutorials)