

DOCUMENTATION

ASSIGNMENT *1*

STUDENT NAME: Butas Rafael-Dorian
GROUP: 30424

CONTENTS

1. Assignment Objective	3
2. Problem Analysis, Modeling, Scenarios, Use Cases.....	4
3. Design	6
4. Implementation	9
5. Results.....	17
6. Conclusions.....	19
7. Bibliography	20

1. Assignment Objective

1.1. Main Objective:

Design and implement a polynomial calculator application that allows users to perform arithmetic and calculus operations on two polynomials.

1.2. Sub-Objectives:

<i>Sub-Objective</i>	<i>Section</i>
<i>Define functional requirements</i>	<i>Problem Analysis, Modeling, Scenarios, Use Cases</i>
<i>Create use cases with diagrams and descriptions</i>	<i>Problem Analysis, Modeling, Scenarios, Use Cases</i>
<i>Design object-oriented programming structure</i>	<i>Design</i>
<i>Create UML package design and class diagram design</i>	<i>Design</i>
<i>Choose the best fitting data structures</i>	<i>Design</i>
<i>Implement the Monomial class used for monomial object manipulation</i>	<i>Implementation</i>
<i>Implement the Polynomial class used for polynomial object(collection of monomials) manipulation</i>	<i>Implementation</i>
<i>Implement the DivisionResults class used for storing the results of the division as a new object containing two polynomial objects</i>	<i>Implementation</i>
<i>Implement the logic for arithmetic and calculus operations on the polynomials</i>	<i>Implementation</i>
<i>Implement the logic for parsing the polynomials from a string to a polynomial object</i>	<i>Implementation</i>
<i>Implement the logic for transforming the result polynomial from a polynomial object to a string (to display it).</i>	<i>Implementation</i>
<i>Implement the user-friendly graphical interface</i>	<i>Implementation</i>
<i>Conduct comprehensive testing using Junit</i>	<i>Results</i>
<i>Draw conclusions and propose future developments</i>	<i>Conclusions</i>

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. *The functional requirements* of the polynomial calculator application describe the core features and functionalities that define its behavior and capabilities. These requirements outline what the application should do to meet the needs of its users. Here's an overview of the functional requirements (in order) for the project:

2.1.1. Input Polynomial Expressions:

- Description: Users are able to input polynomial expressions as strings in the standard mathematical format. The system is parsing the string input into an internal representation in order to perform arithmetic operations.
- Execution Steps:
 - *User accesses the input text field for polynomial expressions.*
 - *User enters the polynomial expressions in the standard mathematical format.*
 - *For derivative and integration, the system needs only the first polynomial, but for the other operations it needs both of them (or else it will display an error saying that the polynomial is empty).*

2.1.2. Perform Arithmetic Operations:

- Description: Users are able to perform arithmetic operations on polynomials, like addition, subtraction, multiplication, division, derivative, and integration.
- Execution Steps:
 - *User selects the desired arithmetic operation from the interface's buttons.*
 - *The system parses the polynomials into an internal representation (as a polynomial object).*
 - *While parsing, the system validates the input expression for correctness. If the input is valid, the system proceeds with the requested operation; otherwise, it displays an error message.*
 - *The system sorts the resulting polynomial from the highest degree to lowest.*
 - *The system stores the result also as a polynomial object (a map).*

2.1.3. Display Results:

- Description: The system displays the results of arithmetic operations in a clear and understandable format(also as a string).
- Execution Steps:
 - *The system formats the result for display (it transforms it back to a string from a polynomial object)*
 - *The system presents the result to the user in the interface in the result label.*

2.1.4. Error Handling:

- Description: The system handles errors gracefully and provide informative error messages to users in case of invalid input or other errors.
- Execution Steps:

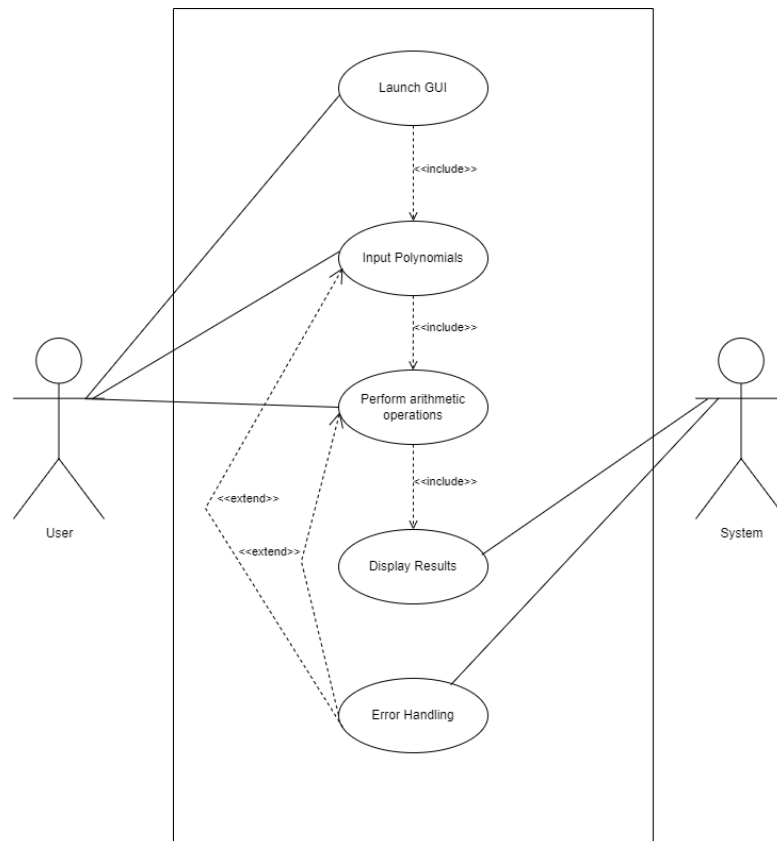
- The system detects an error condition (invalid coefficient, invalid degree, invalid term, arithmetic operation error, empty polynomial, etc.).
- The system generates an appropriate error message explaining the issue.
- The system displays the error message to the user in the interface.

2.1.5. Graphical User Interface (GUI):

- **Description:** The system provides a user-friendly graphical interface for users to interact with and input polynomial expressions.
- **Execution Steps:**
 - User launches the Polynomial Calculator application.
 - The system displays the graphical user interface with input text fields, buttons, labels, and other interactive elements.
 - User interacts with the interface to input polynomial expressions and perform arithmetic operations.
 - System responds to user inputs and displays results or error messages as appropriate.

2.2. Use Case Diagram

A use case diagram is a graphical representation of the interactions between users (actors) and a system, depicting the various use cases or functionalities of the system and how users interact with them. It provides a high-level view of the system's behavior from the perspective of its users. Here's a use case diagram representing the polynomial calculator:



3. Design

The OOP design of the polynomial calculator application promotes modularity, encapsulation, abstraction and maintainability. It adheres to key principles of object-oriented programming, enabling flexibility, extensibility, and robustness in handling polynomial operations and interactions:

3.1. *OOP design:*

The OOP design of the polynomial calculator application promotes modularity, encapsulation, abstraction and maintainability. It adheres to key principles of object-oriented programming, enabling flexibility, extensibility, and robustness in handling polynomial operations and interactions:

3.1.1. Class Organization:

- Monomial: Represents a single term in a polynomial. It encapsulates the degree and coefficient of the term.
- Polynomial: Represents a polynomial as a collection of monomials. It provides methods for adding, removing, and manipulating monomials within the polynomial.
- DivisionResults: Holds the results of polynomial division, including quotient and remainder.
- Operations: Contains methods for performing various operations on polynomials, such as addition, subtraction, multiplication, division, differentiation, and integration, parsing the polynomial(from string to polynomial as a map) and transforming the polynomial(from polynomial object as a map to string).
- App: The main application class responsible for the user interface and interaction.
- OperationsTest: Verifies the correctness of Operations class methods, ensuring they perform polynomial operations accurately and reliably.

3.1.2. Encapsulation:

- Each class encapsulates its data and functionality, exposing only necessary methods for interaction. For example, Monomial encapsulates its degree and coefficient, and Polynomial encapsulates a collection of monomials.

3.1.3. Abstraction:

- The application abstracts polynomial operations into the Operations class, providing a high-level interface for performing operations without exposing the underlying implementation details.

3.1.4. Modularity:

- The application is organized into packages (org.Polynomial, org.Polynomial.operations, Testing), promoting modularity and separation of

concerns. Each package contains related classes, facilitating easier maintenance and understanding of the codebase.

3.1.5. Dependency Injection:

- The Operations class depends on the Polynomial class for performing polynomial operations. This dependency is managed through method calls, enabling loose coupling and dependency injection.

3.2. *UML package*

The UML package diagram provides a high-level overview of the organizational structure of the polynomial calculator application. It shows how classes are grouped into packages based on their functionalities and dependencies. This modular organization helps in maintaining the codebase, promoting code reusability, and managing dependencies effectively. Each package encapsulates related classes, promoting encapsulation and separation of concerns. Moreover, the dependency relationships between packages ensure that classes within different packages can interact as needed, facilitating modular development and testing:

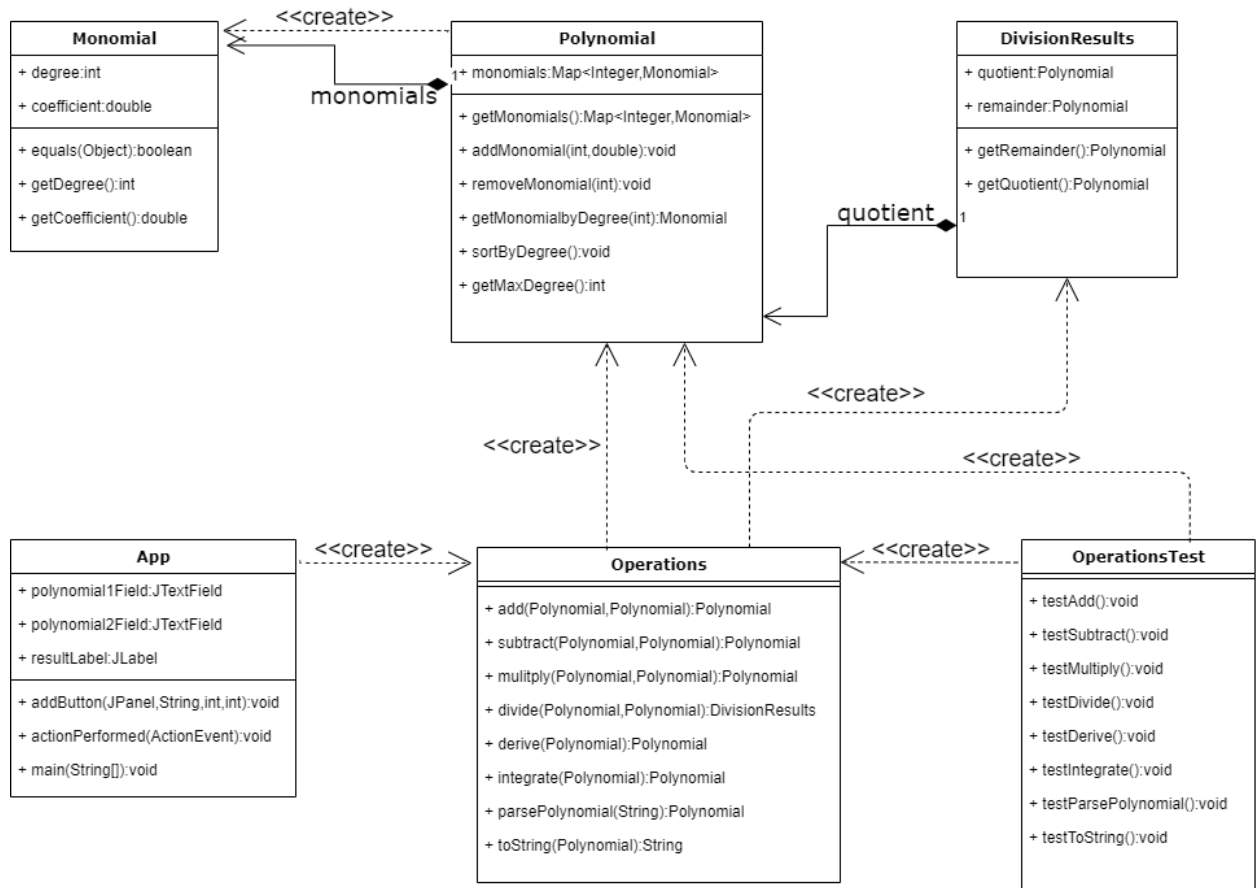
3.2.1. Packages:

- Testing: This package contains test classes for testing the functionalities of the polynomial calculator application.
- org.Polynomial: This package contains classes related to polynomial manipulation.
- org.Polynomial.operations: This package contains classes specifically related to polynomial operations.

3.2.2. Relationships:

- Testing package depends on:
 - org.Polynomial package for testing polynomial-related functionalities.
 - org.Polynomial.operations package for testing polynomial operations.
- org.Polynomial.operations package depends on:
 - org.Polynomial package for accessing classes related to polynomial manipulation.

3.3. *Class Diagram*



3.4. Used data structures

The following data structures are used:

- Map<Integer, Monomial> monomials: This map data structure is used in the **Polynomial** class to store monomials. Monomials are mapped to their degrees, allowing efficient access to monomials by their degrees. This is also a linked hashmap, to keep the same order of the introduced monomials.
- TreeMap<Integer, Monomial> sortedMonomials: This TreeMap data structure is used in the **Polynomial** class to sort monomials based on their degrees. It is utilized in the **sortByDegree()** method to ensure that monomials are ordered by their degrees.
- Map.Entry<Integer, Monomial> entry: This data structure is used in various for-each loops throughout the code to iterate over the entries (key-value pairs) of the monomials map in the **Polynomial** class. It provides a convenient way to access both the degree and monomial objects during iteration.

These data structures are chosen for their suitability in organizing and manipulating polynomial terms efficiently. The use of maps allows for easy lookup and removal of monomials by their degrees, while TreeMap ensures that monomials are sorted by their degrees, facilitating operations like polynomial addition and subtraction.

4. Implementation

In this chapter, a detailed description of each class in the polynomial calculator application will be provided. This includes an overview of their fields, constructors, and methods, as well as their roles and interactions within the application. Understanding each class is essential for comprehending how the application functions and how different components work together to perform polynomial operations. Additionally, the implementation of the graphical user interface (GUI) and how it facilitates user interaction with the calculator will be explored. This examination aims to provide insight into the intricacies of each class and how they contribute to the functionality of the polynomial calculator:

4.1. *Monomial Class:*

4.1.1. Fields:

- degree: Represents the degree of the monomial.
- coefficient: Represents the coefficient of the monomial.

4.1.2. Constructor:

- Monomial(int degree, double coefficient): Initializes a monomial with the given degree and coefficient.

4.1.3. Methods:

- equals(Object obj): Overrides the equals method to check if two monomials are equal based on their degree and coefficient. This method was needed at testing when comparing the monomials of the expected result with the monomial of the actual result (standard equals from assertEquals method failed for monomial objects).
- getDegree(): Returns the degree of the monomial.
- getCoefficient(): Returns the coefficient of the monomial.

```
package org.Polynomial;

33 usages  ▴ Butas Rafael
public class Monomial {
    4 usages
    private int degree;
    4 usages
    private double coefficient;

    1 usage  ▴ Butas Rafael
    public Monomial(int degree, double coefficient) {
        this.degree = degree;
        this.coefficient = coefficient;
    }

    ▴ Butas Rafael
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Monomial other = (Monomial) obj;
        return degree == other.degree && Double.compare(other.coefficient, coefficient) == 0;
    }

    12 usages  ▴ Butas Rafael
    public int getDegree() { return degree; }
    19 usages  ▴ Butas Rafael
    public double getCoefficient() { return coefficient; }
}
```

4.2. Polynomial Class:

4.2.1. Fields:

- monomials: A map storing monomials with their respective degrees as keys.

4.2.2. Constructor:

- Polynomial(): Initializes an empty polynomial.

4.2.3. Methods:

- addMonomial(int degree, double coefficient): Adds a monomial to the polynomial.
- removeMonomial(int degree): Removes a monomial from the polynomial based on its degree.
- sortByDegree(): Sorts the monomials in the polynomial by their degrees by putting them into a treemap with a custom comparator (in reverse order) and then putting them back into a linked hashmap which keeps the order of the insertions. Used to sort the resulting polynomials of each operation.
- getMaxDegree(): Returns the maximum degree of the polynomial by putting the monomial into a normal treemap and taking the last key of it.
- getMonomials(): Returns the map of monomials in the polynomial. Used in for loops to iterate through each element of the monomials map and in testing when comparing the monomials of two polynomial objects using assertEquals.
- getMonomialbyDegree(int degree): Returns the monomial with a specific degree from the polynomial. Used at operations for finding a monomial with the same degree in the second polynomial when iterating through every monomial of the first polynomial. It is also used for finding the monomials of the same degree in the same polynomial (after multiplication or after user input) and merge them.

```
public class Polynomial {
    10 usages
    private Map<Integer, Monomial> monomials;
    33 usages  ▶ Butas Rafael
    public Polynomial() { monomials = new LinkedHashMap<>(); }
    101 usages  ▶ Butas Rafael
    public void addMonomial(int degree, double coefficient) {
        if (coefficient != 0) {
            monomials.put(degree, new Monomial(degree, coefficient));
        }
    }
    3 usages  ▶ Butas Rafael
    public void removeMonomial(int degree) { monomials.remove(degree); }
    10 usages  ▶ Butas Rafael
    public void sortByDegree() {
        TreeMap<Integer, Monomial> sortedMonomials = new TreeMap<>(Comparator.reverseOrder());
        sortedMonomials.putAll(monomials);
        monomials.clear();
        monomials.putAll(sortedMonomials);
    }
    8 usages  ▶ Butas Rafael
    public int getMaxDegree() {
        if (monomials.isEmpty()) {
            return -1;
        }
        TreeMap<Integer, Monomial> sortedMonomials = new TreeMap<>(monomials);
        return sortedMonomials.lastKey();
    }
    30 usages  ▶ Butas Rafael
    public Map<Integer, Monomial> getMonomials() { return monomials; }
    6 usages  ▶ Butas Rafael
    public Monomial getMonomialbyDegree(int degree) { return monomials.get(degree); }
```

4.3. *DivisionResults Class:*

4.3.1. Fields:

- quotient: Represents the quotient polynomial after division.
- remainder: Represents the remainder polynomial after division.

4.3.2. Constructor:

- DivisionResults(Polynomial quotient, Polynomial remainder): Initializes division results with the given quotient and remainder polynomials.

4.3.3. Methods:

- getQuotient(): Returns the quotient polynomial.
- getRemainder(): Returns the remainder polynomial.

```
package org.Polynomial;

8 #sages  ▲ Butas Rafael
public class DivisionResults {

    2 usages
    private Polynomial quotient;

    2 usages
    private Polynomial remainder;

    2 usages  ▲ Butas Rafael
    public DivisionResults(Polynomial quotient, Polynomial remainder)
    {
        this.quotient = quotient;
        this.remainder = remainder;
    }

    3 usages  ▲ Butas Rafael
    public Polynomial getQuotient() { return quotient; }

    3 usages  ▲ Butas Rafael
    public Polynomial getRemainder() { return remainder; }

}
```

4.4. *Operations Class:*

4.4.1. Methods:

- add(Polynomial p1, Polynomial p2): Adds two polynomials using the following logic: for each monomial from the first polynomial, it searches its degree in the second polynomial. If a monomial with the same degree is found, the coefficients are added, and the resulting monomial is created, removing the monomial from the second polynomial. If the system didn't find a monomial with the same degree, the resulting monomial will contain the coefficient of the first one. Finally, all the monomials remained in the second

polynomial (the ones with degrees that were not found in the first polynomial) are added to the resulting polynomial.

```
public Polynomial add(Polynomial p1, Polynomial p2) {
    Polynomial result = new Polynomial();
    for (Map.Entry<Integer, Monomial> entry1 : p1.getMonomials().entrySet()) {
        int degree = entry1.getKey();
        Monomial monomial = entry1.getValue();
        Monomial sameDegree = p2.getMonomialbyDegree(degree);
        if (sameDegree != null) {
            double sumCoefficients = monomial.getCoefficient() + sameDegree.getCoefficient();
            result.addMonomial(degree, sumCoefficients);
            p2.removeMonomial(degree);
        } else {
            result.addMonomial(monomial.getDegree(), monomial.getCoefficient());
        }
    }
    for (Map.Entry<Integer, Monomial> entry2 : p2.getMonomials().entrySet()) {
        Monomial monomial = entry2.getValue();
        result.addMonomial(monomial.getDegree(), monomial.getCoefficient());
    }
    result.sortByDegree();
    return result;
}
```

- subtract(Polynomial p1, Polynomial p2): Subtracts one polynomial from another using the same logic like at addition, but when monomial with same degrees are found, their coefficients are subtracted, not added, in order to create the resulting monomial. Also, at the end, all the monomials of the second polynomial are added to the resulting polynomial with their sign changed.

```
public Polynomial subtract(Polynomial p1, Polynomial p2) {
    Polynomial result = new Polynomial();
    for (Map.Entry<Integer, Monomial> entry1 : p1.getMonomials().entrySet()) {
        int degree = entry1.getKey();
        Monomial monomial = entry1.getValue();
        Monomial sameDegree = p2.getMonomialbyDegree(degree);
        if (sameDegree != null) {
            double diffCoefficients = monomial.getCoefficient() - sameDegree.getCoefficient();
            result.addMonomial(degree, diffCoefficients);
            p2.removeMonomial(degree);
        } else {
            result.addMonomial(monomial.getDegree(), monomial.getCoefficient());
        }
    }
    for (Map.Entry<Integer, Monomial> entry2 : p2.getMonomials().entrySet()) {
        Monomial monomial = entry2.getValue();
        result.addMonomial(monomial.getDegree(), -monomial.getCoefficient());
    }
    result.sortByDegree();
    return result;
}
```

- multiply(Polynomial p1, Polynomial p2): Multiplies two polynomials using the following logic: each monomial from the first polynomial is multiplied with each monomial in the second polynomial (by multiplying their coefficients and adding their powers). Then, it checks if the power of each resulting monomial is already in the resulted polynomial, if it is then their coefficients will be added for that power and putted into the resulting polynomial, if it is not then it is automatically placed in the result alone, as a new power.

```

public Polynomial multiply(Polynomial p1, Polynomial p2) {
    Polynomial result = new Polynomial();
    for (Map.Entry<Integer, Monomial> entry1 : p1.getMonomials().entrySet()) {
        Monomial monomial1 = entry1.getValue();
        for (Map.Entry<Integer, Monomial> entry2 : p2.getMonomials().entrySet()) {
            Monomial monomial2 = entry2.getValue();
            int multiplyDegree = monomial1.getDegree() + monomial2.getDegree();
            double productOfCoefficients = monomial1.getCoefficient() * monomial2.getCoefficient();
            Monomial sameDegree = result.getMonomialByDegree(multiplyDegree);
            if (sameDegree != null) {
                double multiplyCoefficient = productOfCoefficients + sameDegree.getCoefficient();
                result.addMonomial(multiplyDegree, multiplyCoefficient);
            } else {
                result.addMonomial(multiplyDegree, productOfCoefficients);
            }
        }
    }
    result.sortByDegree();
    return result;
}

```

- divide(Polynomial p1, Polynomial p2): Divides one polynomial by another using the following logic: First, it checks if the dividend has a smaller highest degree than the divisor. If it does, then the result of the divide will be 0 as a quotient and the dividend as the remainder. While it doesn't, we divide the highest degree monomial of the dividend with the highest degree monomial of the divisor to obtain the monomial of the quotient. Then, we multiply the obtained term from the quotient with the divisor and we subtract the result of the multiplication from the dividend to obtain the monomial of the remainder.

```

public DivisionResults divide(Polynomial p1, Polynomial p2) {
    Polynomial quotient = new Polynomial();
    Polynomial remainder = new Polynomial();

    for (Map.Entry<Integer, Monomial> entry : p1.getMonomials().entrySet()) {
        Monomial monomial = entry.getValue();
        remainder.addMonomial(monomial.getDegree(), monomial.getCoefficient());
    }

    if (p1.getMaxDegree() < p2.getMaxDegree()) {
        return new DivisionResults(quotient, remainder);
    }

    remainder.sortByDegree();
    p2.sortByDegree();

    while (remainder.getMaxDegree() >= p2.getMaxDegree()) {
        int divisionDegree = remainder.getMaxDegree() - p2.getMaxDegree();
        double divisionCoefficient = remainder.getMonomialByDegree(remainder.getMaxDegree()).getCoefficient() /
            p2.getMonomialByDegree(p2.getMaxDegree()).getCoefficient();
        quotient.addMonomial(divisionDegree, divisionCoefficient);
        Polynomial toMultiply = new Polynomial();
        toMultiply.addMonomial(divisionDegree, divisionCoefficient);
        Polynomial multiplied = multiply(toMultiply, p2);
        remainder = subtract(remainder, multiplied);
    }

    quotient.sortByDegree();
    remainder.sortByDegree();

    return new DivisionResults(quotient, remainder);
}

```

- derive(Polynomial p): Computes the derivative of a polynomial using the following logic: For each monomial, the resulting term will have its degree decreased by one. Also, the coefficient of the resulting term is created by multiplying the input's monomial coefficient with its power.

```
public Polynomial derive(Polynomial p) {
    Polynomial result = new Polynomial();
    for (Map.Entry<Integer, Monomial> entry : p.getMonomials().entrySet()) {
        Monomial monomial = entry.getValue();
        int derivativeDegree = monomial.getDegree() - 1;
        double derivativeCoefficient = monomial.getCoefficient() * monomial.getDegree();
        result.addMonomial(derivativeDegree, derivativeCoefficient);
    }
    result.sortByDegree();
    return result;
}
```

- integrate(Polynomial p): Computes the integral of a polynomial using the following logic: For each monomial, the resulting term will have it's degree increased by one. Also, the coefficient of the resulting term is created by dividing the input's monomial coefficient with the resulting term's power (the input's monomial power+1).

```
public Polynomial integrate(Polynomial p) {
    Polynomial result = new Polynomial();
    for (Map.Entry<Integer, Monomial> entry : p.getMonomials().entrySet()) {
        Monomial monomial = entry.getValue();
        int primitiveDegree = monomial.getDegree() + 1;
        double primitiveCoefficient = monomial.getCoefficient() / (monomial.getDegree() + 1);
        result.addMonomial(primitiveDegree, primitiveCoefficient);
    }
    result.sortByDegree();
    return result;
}
```

- parsePolynomial(String polynomial): Parses a string representation of a polynomial into a polynomial object representation. A regex pattern was used that allowed the system to split the string into groups, each group containing a monomial. The groups were also splitted, taking their degree and coefficient. If the coefficient was not there, depending on the sign (+ or -) it was made 1 or -1. For each matcher found, the system is checking if that degree is already present in the parsed input, and in the case in which it is, it adds the coefficient of the current monomial with the one already present in the parsed input. In the regex pattern, every character was allowed for coefficient, degree, term, etc, but the system is throwing exceptions if it couldn't parse the coefficient and the degree, if x/X is not present as the term of the group, or if the polynomial is empty. Also, any number of spaces anywhere in the input is allowed in the regex pattern and doesn't influence the proper working of the algorithms.

```

    private Polynomial parsePolynomial(String polynomial) {
        Polynomial result = new Polynomial();

        if (polynomial.isEmpty()) {
            throw new IllegalArgumentException("Polynomial is empty");
        }

        Pattern p = Pattern.compile("^((-?)(\\d+\\.?(\\d+)?e(\\d+)|(\\d+\\.?(\\d+)?e(-?)\\d+))$)");

        Matcher m = p.matcher(polynomial);

        while (m.find()) {
            String term = m.group(1);
            if (!term.toLowerCase().contains("e")) {
                throw new IllegalArgumentException("Invalid term: " + term);
            }
            String coefficient = m.group(1).replaceAll("^((-?)", "");
            String degree = m.group(2);
            double resultCoefficient;
            int resultDegree;

            if (coefficient.isEmpty()) {
                resultCoefficient = 1;
            } else if (coefficient.equals("+")) {
                resultCoefficient = 1;
            } else if (coefficient.equals("-")) {
                resultCoefficient = -1;
            } else {
                try {
                    resultCoefficient = Double.parseDouble(coefficient);
                } catch (NumberFormatException e) {
                    throw new IllegalArgumentException("Invalid coefficient: " + coefficient);
                }
            }

            try {
                resultDegree = Integer.parseInt(degree);
            } catch (NumberFormatException e) {
                throw new IllegalArgumentException("Invalid degree: " + degree);
            }

            Monomial sameDegree = result.getMonomialByDegree(resultDegree);

            if (sameDegree != null) {
                result.addMonomial(resultDegree, coefficient + resultCoefficient + sameDegree.getCoefficient());
            } else {
                result.addMonomial(resultDegree, resultCoefficient);
            }
        }

        result.sortByDegree();
        return result;
    }
}

```

- toString(Polynomial p): Converts a polynomial from polynomial object representation to its string representation. The system iterates through each monomial and gets its degree and coefficient. If the coefficient is 0 it continues with the next monomial. If the coefficient is positive (negative), the system appends to the solution the character + (-). A special case is treated here, the one in which the system is dealing with the first monomial. In this case the + is not appended, just the - . If the degree of the monomial is 0 or the coefficient is +1 or -1, the absolute value of the coefficient (1) is appended. If the degree is greater than 0, x is appended, and if its greater than 1, the character ^ is appended to the solution together with the degree. Finally, if the solution is empty, the function returns 0.

```
public String toString(Polynomial p) {
    StringBuilder stringBuilder = new StringBuilder();
    boolean isFirstElement = true;

    for (Map.Entry<Integer, Monomial> entry : p.getMonomials().entrySet()) {
        Monomial monomial = entry.getValue();
        int degree = monomial.getDegree();
        double coefficient = monomial.getCoefficient();

        if (coefficient == 0) {
            continue;
        }

        if (!isFirstElement) {
            if (coefficient > 0) {
                stringBuilder.append("+");
            } else {
                stringBuilder.append("-");
            }
        } else {
            isFirstElement = false;

            if (coefficient < 0) {
                stringBuilder.append("-");
            }
        }

        if (Math.abs(coefficient) != 1 || degree == 0) {
            stringBuilder.append(Math.abs(coefficient));
        }

        if (degree > 0) {
            stringBuilder.append("x");

            if (degree > 1) {
                stringBuilder.append("^").append(degree);
            }
        }

        if (stringBuilder.length() == 0) {
            return "0";
        }

        return stringBuilder.toString();
    }
}
```

4.5. App Class (Graphical User Interface):

The App class is the central component of the polynomial calculator application, responsible for creating the main graphical user interface (GUI) and facilitating user interaction with the calculator's functionalities. It extends JFrame, allowing it to function as the main application window, and implements the ActionListener interface to respond to user actions such as button clicks.

4.5.1. Fields:

- polynomial1Field: A JTextField component where users can input the first polynomial.
- polynomial2Field: A JTextField component where users can input the second polynomial.
- resultLabel: A JLabel component used to display the result of polynomial operations.

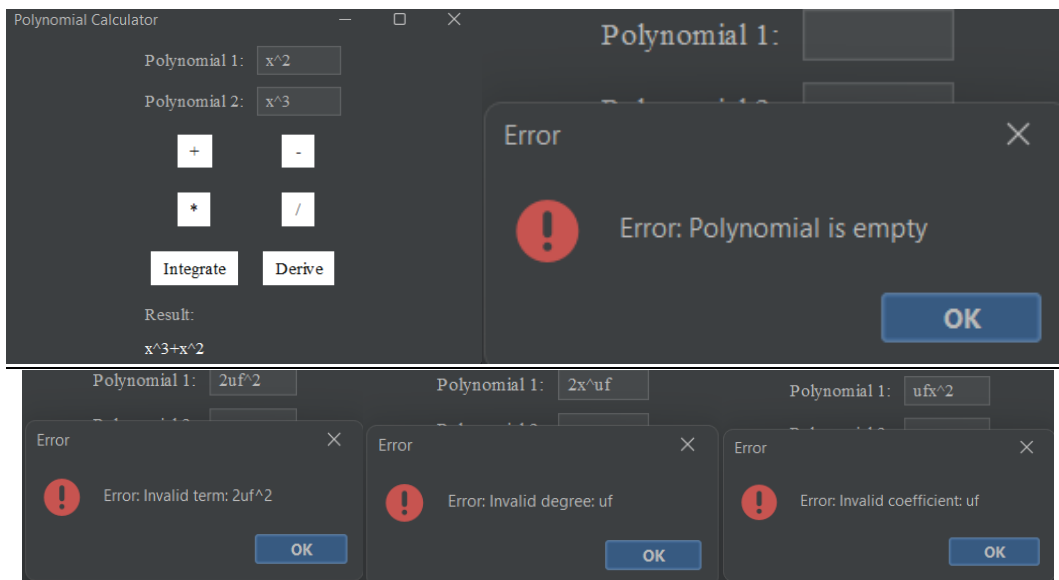
4.5.2. Constructors:

- App (): Initializes the main application window. It sets the title to "Polynomial Calculator", configures the default close operation to exit the application, and sets the preferred dimensions of the window to 400x300 pixels. Additionally, it installs the FlatDarkLaf look and feel for a visually appealing interface. The constructor also sets up the GUI components such as text fields, labels, and buttons within a JPanel using a GridBagLayout. These components are then added to the panel, and the panel is added to

the main frame. Finally, the `pack()` method is called to resize the frame to fit its contents, and the frame is centered on the screen using `setLocationRelativeTo(null)`

4.5.3. Methods:

- `addButton(JPanel panel, String text, int gridX, int gridY)`: This method dynamically creates and adds a button to the specified JPanel with the provided text and position in the grid layout. It configures the button's appearance and behavior, such as setting its background color, font, size, and adding an action listener to handle button clicks.
- `main(String[] args)`: This method serves as the entry point of the application. It ensures that Swing components are accessed and modified on the Event Dispatch Thread (EDT) by invoking the `SwingUtilities.invokeLater()` method. Within this method, an instance of the App class is created, and the application window is made visible to the user.
- `actionPerformed(ActionEvent e)`: This method is invoked whenever a user interacts with the GUI components, such as clicking a button. It handles various actions based on the source of the event (e.g., button clicked). Inside this method, polynomial operations are performed based on user input retrieved and parsed from the text fields (`polynomial1Field` and `polynomial2Field`). These operations are delegated to methods in the Operations class, and the result is transformed back to a string and displayed in the `resultLabel`. If an exception occurs during the operation, an error message is displayed using a `JOptionPane`.



5. Results

In this section, the testing scenarios and results obtained from the JUnit tests will be presented. The JUnit tests are an integral part of ensuring the correctness and robustness of the polynomial calculator application. Each test case verifies specific functionalities of the application, providing assurance of its accuracy and reliability.

5.1. *Testing Scenarios:*

- Addition Operation Test: This test case verifies the correctness of the addition operation performed on two polynomials. It ensures that the addition operation yields the expected result polynomial. Test inputs include polynomials with varying degrees and coefficients to cover different scenarios, including positive, negative, and zero coefficients.
- Subtraction Operation Test: This test case validates the accuracy of the subtraction operation between two polynomials. It confirms that the subtraction operation produces the expected result polynomial. Test inputs include polynomials with diverse terms to assess the handling of different polynomial structures.
- Multiplication Operation Test: This test case checks the correctness of the multiplication operation on two polynomials. It verifies that the multiplication operation generates the anticipated result polynomial. Test inputs encompass polynomials with multiple terms to evaluate the algorithm's scalability and efficiency.
- Division Operation Test: This test case evaluates the division operation between two polynomials, ensuring that both the quotient and remainder are computed correctly. Test inputs cover various scenarios, including cases where the divisor is of higher degree than the dividend, to validate the robustness of the division algorithm.
- Derivative Calculation Test: This test case validates the accuracy of the derivative calculation for a given polynomial. It confirms that the derivative of the polynomial is computed correctly. Test inputs include polynomials with different degrees to assess the differentiation algorithm's correctness and precision.
- Integration Calculation Test: This test case checks the correctness of the integration calculation for a given polynomial. It verifies that the integral of the polynomial is calculated accurately. Test inputs span a range of polynomial expressions to assess the integration algorithm's accuracy and handling of various polynomial structures.
- Polynomial Parsing Test: This test case ensures the proper parsing of polynomial expressions from string format to polynomial objects. It confirms that valid polynomial expressions are parsed correctly while handling invalid expressions with appropriate

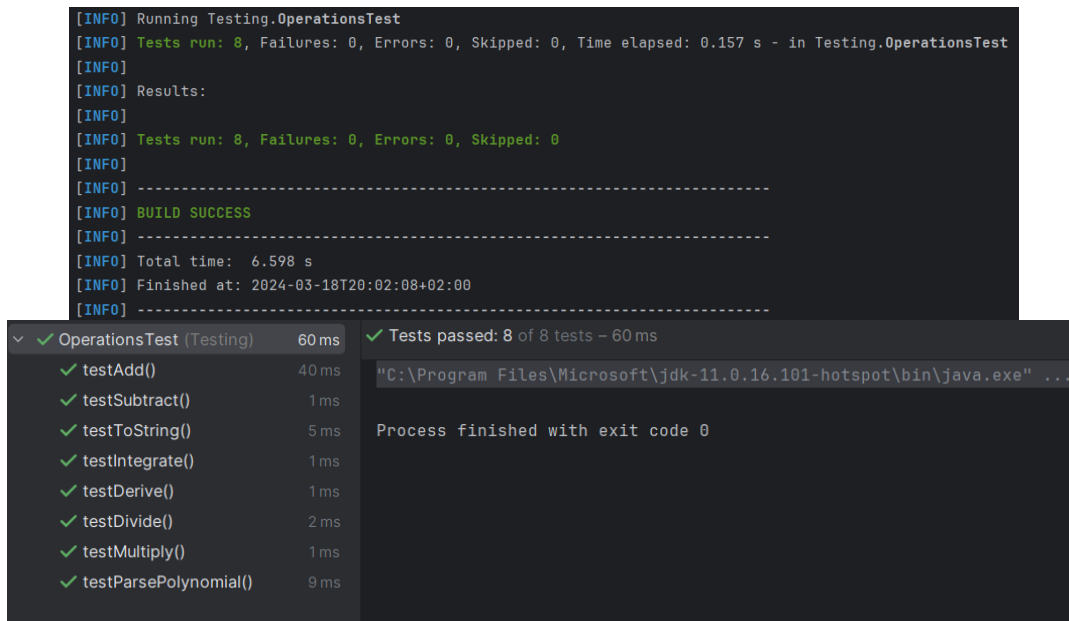
exceptions. Test inputs include valid and invalid polynomial strings to assess the robustness and error-handling capabilities of the parsing algorithm.

- **Polynomial to String Conversion Test:** This test case validates the conversion of polynomial objects to string representations. It ensures that the string representation accurately reflects the polynomial expression. Test inputs cover polynomials with different degrees and coefficients to evaluate the formatting and precision of the string conversion algorithm.

5.2. *Results Summary:*

- All test cases passed successfully, demonstrating the correctness and reliability of the polynomial calculator application.
- The application's functionalities, including addition, subtraction, multiplication, division, derivative calculation, and integration, are thoroughly tested and validated under diverse scenarios.
- Error handling mechanisms for parsing invalid polynomial expressions are effective, as indicated by the appropriate exceptions thrown during testing.
- The graphical user interface (GUI) of the application, although not directly tested through JUnit, was manually tested to ensure proper functionality and user-friendliness.

These detailed test results provide comprehensive insight into the functionality and accuracy of the polynomial calculator application, instilling confidence in its performance and adherence to the assignment objectives.



```
[INFO] Running Testing.OperationsTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.157 s - in Testing.OperationsTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.598 s
[INFO] Finished at: 2024-03-18T20:02:08+02:00
[INFO] -----
```

Test Case	Duration
✓ OperationsTest (Testing)	60 ms
✓ testAdd()	40 ms
✓ testSubtract()	1 ms
✓ testToString()	5 ms
✓ testIntegrate()	1 ms
✓ testDerive()	1 ms
✓ testDivide()	2 ms
✓ testMultiply()	1 ms
✓ testParsePolynomial()	9 ms

✓ Tests passed: 8 of 8 tests – 60 ms

"C:\Program Files\Microsoft\jdk-11.0.16.101-hotspot\bin\java.exe" ...

Process finished with exit code 0

6. Conclusions

In conclusion, the development of the polynomial calculator application has been a valuable learning experience, providing insights into various aspects of software design, implementation, and testing. The following conclusions can be drawn from the assignment:

- Achievement of Objectives: The main objective of designing and implementing a polynomial calculator application has been successfully accomplished. The application allows users to perform arithmetic and calculus operations on polynomials accurately and efficiently.
- Understanding of Polynomial Operations: Through the implementation process, a deeper understanding of polynomial operations, including addition, subtraction, multiplication, division, differentiation, and integration, has been gained. Handling complex mathematical operations programmatically has enhanced problem-solving skills.
- Object-Oriented Design Principles: The application's design adheres to key principles of object-oriented programming (OOP), such as encapsulation, abstraction, and modularity. This approach promotes code reusability, maintainability, and scalability, contributing to a robust and flexible codebase.
- Testing and Error Handling: Comprehensive testing, including JUnit tests, has been conducted to ensure the correctness and reliability of the application. Effective error handling mechanisms have been implemented to provide informative error messages and gracefully handle invalid inputs.
- User Interface Design: The graphical user interface (GUI) of the application provides a user-friendly environment for interacting with polynomial expressions. Attention to usability and aesthetics enhances the overall user experience.

6.1. *Future Developments:*

- Enhanced Functionality: Future developments may focus on expanding the functionality of the polynomial calculator application to include advanced features such as symbolic computation, graph plotting, and support for more complex mathematical expressions.
- Optimization and Performance: Optimization efforts can be undertaken to improve the efficiency and performance of polynomial operations, particularly for large polynomial expressions. Implementing algorithms for optimized polynomial manipulation can enhance computational speed and resource utilization.

- User Feedback Incorporation: Gathering user feedback and incorporating suggestions for interface improvements and feature enhancements can further enhance the usability and effectiveness of the application. Iterative development cycles based on user input can lead to continuous improvement.
- Cross-Platform Compatibility: Consideration may be given to making the application compatible with a wider range of platforms and devices, including web browsers, mobile devices, and desktop environments. This can broaden the reach and accessibility of the application to a larger user base.

In summary, the polynomial calculator application represents a successful implementation of mathematical concepts into a practical software tool. Continued refinement and expansion of the application can further enrich its capabilities and utility.

7. Bibliography

The following references were consulted during the implementation of the polynomial calculator application:

1. Swing: <https://docs.oracle.com/javase/tutorial/uiswing/index.html>
2. Junit: <https://www.baeldung.com/junit-5>
3. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S1.pdf
4. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S2.pdf
5. https://dsrl.eu/courses/pt/materials/PT_2024_A1_S3.pdf
6. <https://gitlab.com/dsrl-pt/pt-gui-demo-swing.git>
7. <https://www.javatpoint.com/java-regex>